ASSEMBLY LANGUAGE DEBUGGING HARDWARE/SOFTWARE FOR THE COMMODORE 64 COMPUTER

By

Charles Raymond Jacobs
B.S., University of Louisville, 1984

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed Scientific School
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Electrical Engineering

August 1987

ASSEMBLY LANGUAGE
DEBUGGING HARDWARE/SOFTWARE
FOR THE COMMODORE 64 COMPUTER

Submitted by: Charles R. Jacobs
Charles R. Jacobs

A Thesis Approved On

JULY 24, 1987
Date

by the Following Reading and Examination Committee:

[illegible]
Thesis Director, Doctor William H. Pierce

[illegible]
Professor Kenneth E. Stoll

[illegible]
Doctor Thomas G. Cleaver

ACKNOWLEDGEMENTS

The author would like to express his sincere appreciation to the thesis director, Dr. William H. Pierce, for all his assistance and patience throughout this project. He would also like to acknowledge and thank Professors Kenneth Stoll and Thomas Cleaver for serving as review committee members.

Finally, the author would like to thank the Management of Louisville Gas and Electric Company for providing the author with the opportunity to complete his thesis research while being under their employment.

Without all the above mentioned individuals, this project could not have been successfully completed.

ABSTRACT

This thesis project involves the design, construction, and implementation of the hardware and software necessary to create a low cost assembly language debugging utility. The utility is designed specifically for the Commodore 64 home computer.

The hardware consists of a software controlled counter which generates a non-maskable interrupt (NMI) after a predetermined count. The interrupt occurs once during each instruction of the target program. The above mentioned process is known as single stepping with a timed interrupt.

The software consists of a consolidation of commands found in high level language debuggers and microprocessor development systems. There are also several innovative commands such as subroutine suppression and multiple execution of breakpoints which were developed by the author.

The above mentioned hardware and software was sucessfully implemented. The debugging system has proved extremely useful for the author.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF ILLUSTRATIONS

I. INTRODUCTION

The microprocessor development system is a tremendously important tool to the engineer or designer. As part of the development system, the software debugging utility is equally important.

A debugging utility can be defined as a group of individual command routines, operating under a centralized monitoring program, which allow the user to monitor and control the execution of the target code. This monitoring program is known as the debug monitor. The various command routines should be specifically written for the host computer system, thus taking advantage of the computer's good points while supplementing the system's deficiencies. Examples of these routines are assembly, disassembly, memory/register display/modification, monitoring memory/registers for a predetermined value, setting breakpoints, and single stepping. Although each of the above mentioned commands has great importance when attempting to debug faulty object code, the most important debugging tool would have to be single stepping.

Single stepping can be implemented using various methods. This is dependent on the type of microprocessor utilized. Most modern 16-bit microprocessors support single stepping using a purely software approach (see Chapter II). All of the 8-bit microprocessors studied by the author require an externally generated interrupt which causes a control transfer to the single step routine. The interrupt can originate from elsewhere in the microcomputer system, or from a completely external hardware circuit.

The host computer system chosen for the debugging hardware/software implementation was the Commodore 64. This computer utilizes the 6510 microprocessor which belongs to the more familiar 6502 family of microprocessors. Since the Commodore 64 lacks the necessary hardware to single step its CPU, custom hardware has been added (see Chapter IV).

Previous work on this subject includes thesis research by Kamran Khosrani-Kamrani entitled "<u>Toward a Low Cost Microprocessor Development System</u>". Kamrani discusses various topics related to the hardware aspect of microprocessor development systems. As a host system, he also chose the Commodore 64. T.P. Hughs and D.H. Sawin III, in their paper "<u>Breakpoint Design for Debugging Software</u>" discuss in great detail the interaction of break points within a debug monitor program. Eugene Zumchak in "<u>Microprocessor Design and Troubleshooting</u>" and R. Bywater in "<u>Hardware/Software Design of Digital Systems</u>" both cover the hardware and software aspects of microprocessor development and debugging. All of the above mentioned material as well as others can be found in the Bibliography.

Based on the previous work as well as other contemporary sources, the author has created his own customized software debugging system. This system incorporates several features of other debuggers, as well as some innovative commands developed by the author. The major emphasis is on break point implementation and single stepping.

The thesis manuscript is organized as follows: The first section, being this introduction, will attempt to introduce the reader to the subject matter and direct him or her to specific points of interest in the text. The second chapter gives a brief analysis of how the most common debugging techniques are implemented. It will also give an insight into what commands and/or techniques are important in a debugging utility. The third chapter deals with the operations of the Commodore 64 microcomputer. It is the Commodore 64 which the author has chosen to implement the prototype debugger system. The fourth chapter encompasses the design of the prototype hardware. The hardware is necessary in order to generate a timed non-maskable interrupt which initiates the single stepping process. Chapter V describes the design and implementation of the prototype debugging software. Chapter VI provides a conclusion and offers several recommendations to the research and prototyped system.

II. SOFTWARE DEBUGGING

This chapter's purpose is to provide a tutorial and review of software debugging and to provide a philosophical foundation for the thesis hardware of Chapter IV and the thesis software of Chapter V.

The analysis of software debugging can be broken down into three major divisions. The first is simulation and debugging. Although the terms have different meanings, they both require the same type of monitoring commands. The second section explains how the two most common types of debugging commands are implemented on various microprocessors. The final division contains what the author feels is the "ideal" debugging system.

A. Simulation and Debugging

Software development is one major aspect in overall microprocessor system development. Once the hardware is determined to be reasonably correct, the operation of the microprocessor system depends on its software. The reliability of this software must be tested and ensured. The two systematic methods of testing software are simulation and debugging. These methods are closely related and require further definition.

Simulation, in the sense of a software development system, can be defined as the setting up of conditions for which the system operation is observed. This is accomplished by halting execution during key program phases and altering the microprocessor registers or computer memory locations. The conditions mentioned above can be either internal or external in nature. An internal condition is one that originates from within the microprocessor system such as program flags or stored data. An external condition is one that originates from outside the system such as data bytes obtained through an I/O port or external bus connection. By software simulation, the actual hardware connection does not need to be completed in order to examine the result. Also segments of programs or subroutines can be tested without execution of the calling routines.

Debugging can be defined as the removal of errors within a system. Although debugging connotes the presence of a "bug" in the programming, debugging techniques can also be used throughout the software development stage to ensure the proper operation the the program segments. In addition to the removal of outright errors, the debugging process can also aid in the optimization of the code.

Software debugging may or may not involve direct evaluation of the operation of the object code. When the source program is written in assembly language, debugging always involves the object code. When the source program is written in a high level language and compiled to machine language object code, it may or may not be appropriate to examine the object code for debugging purposes. In systems programming, especially microprocessor development or compiler development, the object code must be examined frequently even when it comes from a high level language. This thesis is specifically oriented to those applications whose nature does require direct examination of the object code.

B. Debugging Techniques

The two most important aspects of software debugging that will be addressed in this paper are single stepping and break point design.

1. Single Stepping

Single stepping involves suspending program execution after each instruction, storing the microprocessor registers, examining the register and memory contents, and then restoring the registers. The single step process repeats itself until the loop is terminated by the monitor program. The single step routine must be totally transparent to the target program. This means that all microprocessor flags and register values must be returned to their previous value before the next target instruction is encountered.

The most obvious use of single stepping is for tracing program execution. For this purpose, after each instruction, the register values along with the current address and mnemonic representation of the instruction are displayed. Single stepping provides the data needed to "home in" on a problem. It can determine, for example, whether a routine fails because it is fed the wrong data, or because it is structurally defective. Once the defective location is found, single stepping through the actual error process usually supplies the definitive insight to correct the problem.

There are other uses of single stepping where a display is not necessary. Single stepping can be used to implement watch commands. A watch command instructs the single step routine to monitor a certain register or memory location for a predetermined value. This comparison is performed after each target program instruction. When the predetermined value is detected, the execution of the target program is halted. Single stepping through each instruction gives the greatest control, but it also introduces a large time delay. It is not the only design option. W.H. Pierce is currently experimenting with an "Occasional" stepping for debugging purposes, and finds it useful[1]. A third use of single stepping is to set break points in read only type memory (break points in read/write memory are usually implemented by over-writes). This application is covered in detail, along with break points, in part 2 of this section. Finally, single stepping can be used to monitor the target program so that a specific operation can be carried out when a particular instruction is encountered. A typical application of this would be to execute the target program until a jump or branching instruction is encountered.

In general, the transfer of control from the target program to the single stepping routine can be implemented using two different methods, interrupts or internal microprocessor flags. The single stepping method is dependent upon the type of microprocessor utilized. Table I (page 9) summarizes the levels of debugging support implemented by various microprocessors. The information contained within the table was compiled by the author from Osborne's "4 & 8 Bit Microprocessor Handbook" and Osborne's "16 Bit Microprocessor Handbook" (see bibliography). As observed in Table I, none of the eight bit microprocessors examined by the author possessed the ability to single step by internal flags. Greater hardware support for debugging appears in the 16 bit microprocessors and will be discussed later. Single stepping of the eight bit machines must be accomplished by means of externally generated interrupts. The most reliable interrupt for this purpose is the non-maskable interrupt (NMI). The NMI must occur once during each target program instruction cycle. By the inherent design of the interrupt process, the current target program instruction will be executed completely and then control will be transferred to the NMI routine. A return from interrupt (RTI) will return the microprocessor control to the next instruction of the target program.

TABLE I

SUMMARY OF MICROPROCESSOR SUPPORT FOR DEBUGGING PURPOSES

Micro-processor	# Ext Bits	Manu-facturer	SYNC	Built in Single stepping	X-fer on Internal Error	Soft-ware Intrpt
3870/F8	8	Fairchild	no	no	no	no
8080A	8	Intel	yes	no	no	yes
8085A	8	Intel	yes(1)	no	no	yes
Z80	8	Zilog	no	no	no	yes
6800	8	Motorola	no	no	no	yes
6502	8	MOS	yes	no	no	yes
6510	8	MOS	no	no	no	yes
2650	8	Signetics	no	no	no	no
TMS9900	16	Texas Ins	yes	no	no	no
8086	16	Intel	no	yes(2)	yes	yes
Z8000	16	Zilog	no	yes(3)	yes	yes
68000	16(5)	Motorola	no	yes(4)	yes	yes

Notes:

(1) SYNC signal can be synthesized by the logical combination of S0 and S1 (bus status indicators).

(2) Single step mode implemented via software, TRAP flag internal to microprocessor.

(3) Single step mode implemented via hardware, STOP signal externally input to microprocessor.

(4) Single step mode implemented via software, TRACE flag internal to microprocessor.

(5) 32 bit internal architecture, 16 externally connected bits.

Two possible origins of the NMI are illustrated as follows. Both circuits were suggested by Eugene Zumchak[2].

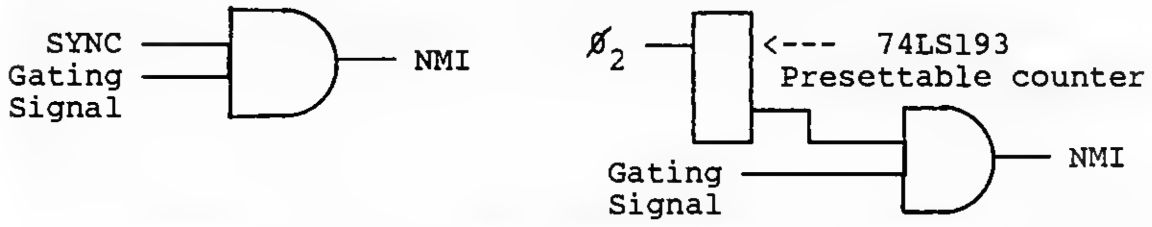


Figure 1 - Two Possible Methods of Generating Interrupts for Single Stepping

The first circuit configuration depends on the presence of the SYNC signal from the microprocessor. In general, the SYNC signal becomes active during each instruction fetch, indicating the beginning of the instruction cycle. The gating signal is software controlled through some type of I/O port or additional external hardware. Its purpose is to prevent unwanted interrupts (single stepping) during parts of the target program that are to be executed normally, and also to prevent interrupts during the single stepping routine itself. The logical ANDing (or NANDing, depending on the required logic levels) of these two signals produce the required NMI. A chip select (CS) signal may be required as an additional input to the NAND gate on some CPUs.

The second circuit configuration must be used when a SYNC signal is not available on the microprocessor. Without a SYNC, there is no external way to determine the beginning of a new instruction cycle. In this circuit a presettable counter is used. Starting at a predetermined value, the counter circuit counts down the system clock pulses. An interrupt is generated when the counter reaches zero. The counter is activated via the software several instructions before the next target program instruction. The counter's initial value must take into consideration the length of the delay before the next target program instruction is encountered. Figure 2 provides a simplified example of this delay. (The example is written using QK-02 addressing nomenclature which is presented in Chapter III, section B3). This example identifies the worst possible case; a NOP being the next target program instruction. The NOP instruction is the shortest instruction, only two clock cycles in length (for the 65xx family of microprocessors). The interrupt, therefore, must occur sometime during the first <u>two</u> clock

```
LDA-- $0330    ! Restore accumulator with the
               !   value before the interrupt.
LDX-- $0331    ! Restore Y-register with the value
               !   before the interrupt.
LDX#  $01      ! Load X-register with $01.
STX-- $DD01    ! Send gating signal to I/O port and
               !   start coutner.
LDX-- $0330    ! Restore X-register to the value before
               !   the interrupt. (counter is counting)
RTI            ! Dummy return from interrupt used to
 .             !   restore the processor status word
 .             !   and load the address of the next
 .             !   target program instruction from the
 .             !   stack.  (counter is counting)
NOP            ! This would be the next instruction in
               !   the target program.  The counter
               !   will reach zero and generate an
               !   interrupt during this instruction
               !   and the single step processing will
               !   begin.
```

Figure 2 - Typical Program Sequence for Counter Generated Interrupt.

cycles of the target instruction - regardless of the instruction or its length. Via the NMI, the single step data processing takes place. It can be seen that the timing for this type of single step scheme is critical. Again, as in the first circuit, the gate signal is software controlled to prevent unwanted interrupts.

The 16 bit microprocessors support single stepping is a more straightforward way. The Z8000 (Zilog) has a dedicated input signal to the microprocessor referred to as STOP. When this signal is brought low, the microprocessor enters a single step mode. After each instruction is executed program control is passed to the single stepping routine. This is accomplished through the use of a special vector known as the STOP vector. The STOP signal of the Z8000 must be software controlled using a gating signal with origins similar to the gate signal mentioned for the eight bit machines.

Both the 8086 (Intel) and the 68000 (Motorola) use a purely software approach to single stepping. The 8086 has an internal microprocessor flag called TRAP. When set, program control is transferred to a TRAP routine after one instruction is executed. The flag automatically resets itself after the instruction is executed. Therefor the TRAP flag must be reset at the exit of the single step (TRAP) routine. The 68000 utilizes a flag called TRACE. Its operation is identical to that of the TRAP flag in the 8086. Routines similar to the TRAP and TRACE routines but called upon by different vectors, can also be used to allow the microprocessor to recover from fatal errors such as divide by zero and attempting to execute invalid or undefined opcodes.

Regardless of the type of microprocessor used (either 8 or 16 bit) the target program execution speed is greatly reduced during single stepping. This is due to the fact that the microprocessor control is diverted to the single step routine after each target program instruction is executed. The reduction can be as large as 2,000 times slower than the normal execution speed. This, of course, is dependent on the complexity of the single step routine. If a screen display must be provided, the speed reduction is not significant since even a larger delay (through a timing loop) will be needed to allow the user time to read and analyze the screen. If single stepping is being implemented to facilitate a watch command or ROM break point (ie. no screen display) then the overall execution speed becomes a critical issue. A trade off point between the amount of data processing in the single step routine and the tolerable speed reduction during target program execution must be determined.

2. Break Points

The second major topic of software debugging is setting break points. Break points are used to stop target program execution at a predetermined address location. During a break routine, the break address and the micro-processor registers are usually displayed. When a break is encountered, all register values must be stored so that execution can be continued if the user wishes.

Break points have several applications. For simulation purposes, once the target program is stopped, the microprocessor registers and memory locations can be altered. When program execution is continued, these new values are loaded and used within the target program. By setting a break point at the end of a previously debugged routine, the code up until that point can be executed at full microprocessor speed. After the break point, single stepping or other debug monitoring commands can be issued. Break points can also be used to examine the results of a particular routine after it has been executed.

In a more advanced debugging technique, a loop may fail not on its first or second execution, but on its n-th execution. In such cases it is very helpful to activate a break point only after n-1 successful executions.

The setting of a break point can be accomplished in two different ways. The simplest approach is to physically insert a software interrupt instruction at the desired address location in the target program code (over-write). Table I (page 9) summarizes the software type interrupts of various 8 and 16 bit microprocessors. The software interrupt instruction is termed SWI, BRK or RST depending on the type of microprocessor system used (for simplicity, the instruction will be referred to as "BRK"). When target program control reaches the BRK, control is transferred to the break handling routine via an interrupt vector. In all the microprocessors reviewed by the author, except the

6510, the BRK instruction is of the non-maskable type. This insures that the target program cannot inadvertently mask out the break point.

The above mentioned scenario assumes that the target program is located in random access memory (RAM). This is usually the case for programs early in the development stage. Toward the end of a design project, the software may already be committed to read only memory (ROM, PROM, or EPROM). If this is the case, a BRK instruction cannot be written into the target program code. A second approach is to use a microprocessor single step routine. In order to implement this type break point, the target program must be halted after each instruction and a comparison made between the program counter value and the desired break point address. If the values are equal, control is transferred to the break point handling routine. Otherwise, the next instruction of the target program is executed and the process repeats itself.

The first method of setting break points allows full speed execution of the target program. It is therefor the fastest and the most desirable choice. The second method is limited by the execution speed of the single step routine, as discussed earlier.

C. The "Ideal" Debugging Utility

The "ideal" debugging utility, as envisioned by the author, should encompass the following commands and/or features:

1) Setting of at least two independent break points.
2) Allowing break points to be encountered up to 2^n-1 times before breaking execution at it, where n is the word length.
3) Single stepping with one line output on the screen for each instruction (tracing).
4) Controlling the speed of the screen display.
5) Single stepping without the screen display.
6) Suppressing the display of subroutines while in the tracing mode. (executing an entire subroutine in a single step)
7) Full speed, unmonitored execution of the target programs.
8) Setting watch commands for the microprocessor registers as well as individual memory bytes and memory words (two byte combinations).
9) Observe/alter memory at a specific location.
10) Observe/alter the microprocessor registers upon request.

In addition, it is helpful to have a relocatable load command, save memory command and disassembly.

It is the intent of the author to demonstrate that the high level operations, such as those listed above, can be implemented on a low cost hobby or home computer. The computer selected for this project is the Commodore 64. Because the Commodore 64 is considered a hobby computer, little emphasis has been placed on the use of this computer for the simulation and debugging of assembly language programs*. Although several debugging monitors exist, Micromon, Supermon and Hesmon to name a few, none of these possess all the high level commands and features stated above.

In order to implement these commands and features, the internal hardware of the Commodore 64 must be analyzed. This posses some problems since the computer utilizes several proprietary integrated circuits. Fortunately, the 6510 microprocessor belongs to the 6502 family of microprocessors, which are well documented.

Secondly, the Commodore's internal timing and system interaction must be examined. The 6510 will be utilized as the microprocessor for both the target program and the debug monitor program. The timing is critical since the 6510 does not possess an external SYNC. The single stepping routine will be implemented using timed NMIs.

The machine language instruction set associated with the microprocessor must also be fully understood. This is necessary since the target instruction's control over the internal 6510 registers and stack must be propagated through the interrupt. The interrupt routine cannot be allowed to disrupt any of the microprocessor registers if the interrupt is to remain transparent.

Finally, but most importantly, an understanding of

*The earlier Atari computers utilizing the 6502 CPU did include the necessary hardware for single stepping.

the common programming errors which may occur is warranted. Many common errors can be caught during initial assembly of the target program. Examples of these are: Mnemonic/extra byte mismatch, missing operands, and general typographical errors associated with program entry. Other errors such as infinite loops, writing over program RAM, improper branching, or any unexpected program results require high level debugging techniques as mentioned earlier.

III. COMMODORE 64 ARCHITECTURE

The analysis of the Commodore 64 can be broken down into 3 basic sections, the microprocessor, the memory, and the I/O devices. The latter two items will be discussed immediately, the microprocessor has been reserved for section 'B'.

A. System Configuration

1. Memory

The Commodore 64, as its name implies, contains 64k bytes of RAM. The memory is formed by a matrix of eight 64k x 1 bit dynamic RAMs. In addition to the RAM, there are also 20k bytes of ROM memory. This consists of; 8k of Basic interpreter, 8k of Kernal (operating system), and 4k of character ROM. Since the 6510 microprocessor has only 16 address lines, it can only directly access 64k of memory at any one time. The memory management is performed by a programmable logic array (PLA). This device generates the required enable signals which switch the various RAM, ROM and I/O devices in and out of the memory map. Figure 3 (page 20) shows the universal and normal (Basic) memory maps. The inputs to the PLA, which control the memory map, originate from either the microprocessor or the expansion port. A summary of the control signals available at the expansion port can be found in Table II (page 22).

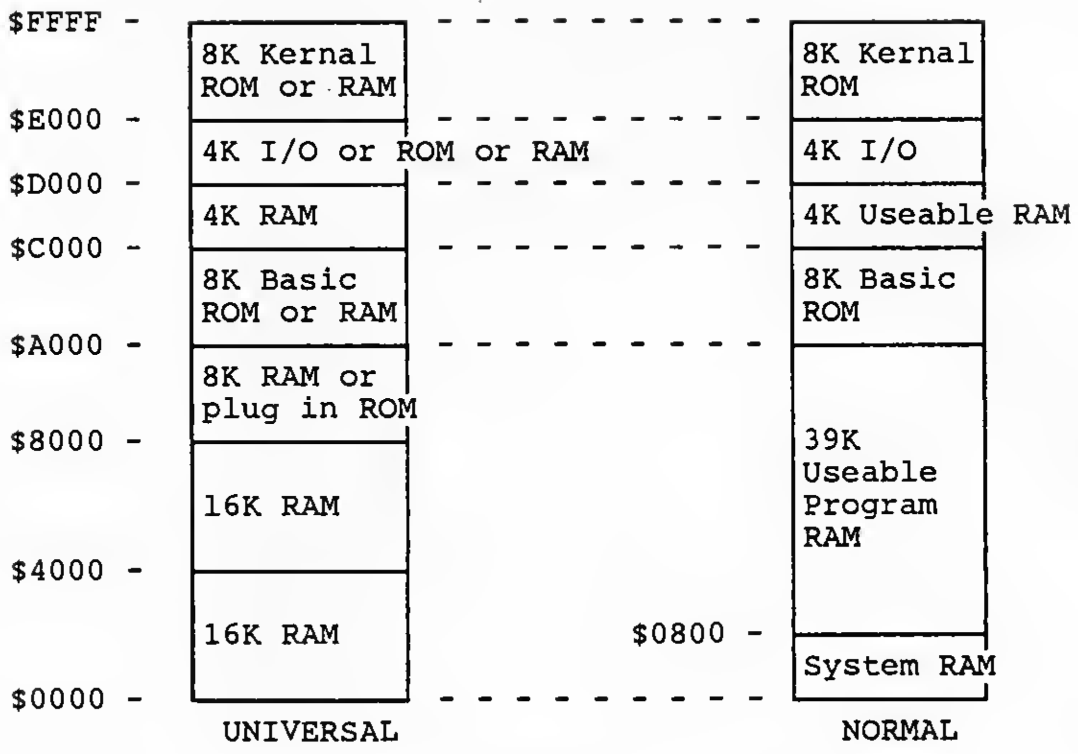


Figure 3 - Commodore 64 Memory Maps

2. I/O Devices

The Commodore 64 has several integrated circuits dedicated to the input and output of data. Two of these ICs have great importance with respect to the system control and timing. These are the video interface controller (VIC) and the complex interface adapters (CIA). Although they are not devices per se, the I/O ports also have great importance to the system since they govern how easily the outside world can be interfaced to the computer. All three items will be discussed following.

The VIC chip's main function is to control the video output. It does this by fetching character values from the screen memory matrix and converting them to their dot representation via the character ROM. After the output of the VIC chip is buffered, it is directly feed into the rf modulator (for television) and to the audio/video port (for monitor).

The character fetches performed by the VIC are normally invisible to the microprocessor since it accesses the data and address bus during $\emptyset_1$ clock cycles only. Sharing the control of the address bus with the microprocessor is accomplished by use of a special control signal generated by the VIC chip. This signal, called Address Enable Control (AEC), causes the address bus connection within the 6510 microprocessor to enter a high impedance state. Some VIC operations require the use of the address bus longer than one $\emptyset_1$ access. The AEC in conjunction with the RDY signal allow the VIC chip to force the microprocessor into a wait state while the VIC performs both $\emptyset_1$ and $\emptyset_2$ bus accesses. The system timing will be discussed in detail in Part B of this section.

A secondary function of the VIC chip is to refresh the dynamic RAMs. This is performed during video raster scans and it is totally transparent to the 6510 microprocessor system.

Another important integrated circuit is the complex interface adapter. The CIAs (two of them) operate much the same as a programmable interface adapter (PIA) and an asynchronous communications interface adapter (ACIA). It allows the microprocessor to read and write the signals required to evaluate the keyboard and control ports. In addition, the CIA converts the parallel information on the data bus into a serial data format for direct communication with peripheral devices. Another feature built into each CIA are various timers which can be programmed and read as time of day clocks.

As mentioned earlier, the Commodore 64 has various ports for connection of auxiliary and peripheral devices. There are dedicated purpose ports such as control ports 1 and 2 (for joystick or paddle), a cassette port, a serial port (for printer or disk drives) and a monitor port. There are also two general purpose ports, the user port and expansion port. The various signals available on the latter two ports are summarized in Table II.

TABLE II
SUMMARY OF CONTROL SIGNALS AVAILABLE ON COMMODORE 64 PORTS.

USER PORT		EXPANSION PORT	
ATN	Connected to CIA 2	A_0-A_{15}	Connected to micro-processor
CND1, CNT2		D_0-D_7	
SP1, SP2		NMI	
PC2		IRQ	
PA2		R/W	
Pb_0-Pb_7		BA	
FLAG2		DMA	
RESET	-- to system	$Ø_2$	-- to system
		DOT CLOCK	-- to system
		EX ROM	Connected to PLA
		GAME	
		ROM L, ROM H	
		I/O_1, I/O_2	

The user port is primarily wired to CIA 2. Through this parallel port the connection of a Centronics or RS-232 interface circuit can be implemented. The user port can also used for various speciality control circuits. The expansion port is normally used for the connection of ROM cartridges. It can also be used for direct memory interfacing applications. The CIA have several of functions dedicated to system operation, and cannot be totally reasigned new I/O tasks.

B. 6510 Microprocessor

The 6510 is an eight bit microprocessor employing N-channel MOS circuit technology. With sixteen address lines, it is capable of directly addressing a full 64K of memory. The principle manufacturer of the chip is MOS Technology Inc. It was introduced in June of 1983, when the Commodore 64 computer entered the market. To the best of

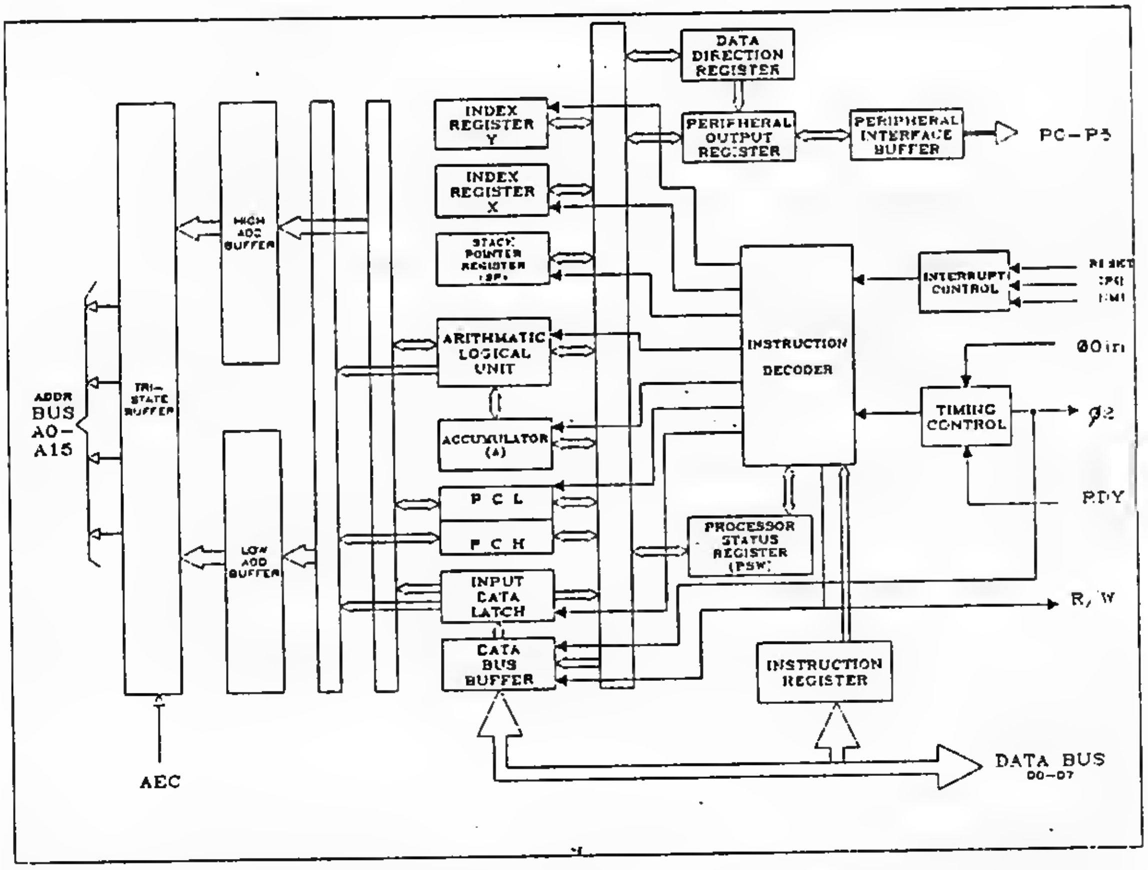


Figure 4 - Block Diagram of the 6510 Microprocessor (Reprinted from the C-64 Programmers Reference Guide, with corrections by the author)

the author's knowledge, this is the only application of the microprocessor. Figure 4 (previous page) shows the internal configuration of the 6510.

Other members of the 65xx microprocessor family include the better known 6502 microprocessor (used in Atari and early 8-bit Apple computers).

The analysis of the 6510 microprocessor can be divided into three parts, a comparison to the more familiar 6502 microprocessor, the timing and control of the 6510, and the software instruction set.

1. Comparison Between the 6510 and the 6502 microprocessor.

There are several differences in the hardware of the 6510 and 6502. These will be discussed in detail in the following paragraphs.

The first major difference between the 6510 and the 6502 is the origin of the system clocks. Both microprocessor systems require two non-overlapping clocks, referred to in the timing diagrams as $\emptyset_1$ and $\emptyset_2$. In the case of the 6502, all that is required is an external crystal or clock signal connected to the $\emptyset_0$ input. The 6502 synthesizes both the $\emptyset_1$ and $\emptyset_2$ clocks from this reference frequency. The 6510, on the other hand, requires the $\emptyset_1$ clock signal to be input (calling it $\emptyset_{0in}$). From the $\emptyset_1$ reference clock, the 6510 synthesizes the $\emptyset_2$ clock output*.

*This is contrary to what is found in the C-64 Programmer`s Reference Guide.[3] In it, the writers claim the $\emptyset_2$ clock is input to the CPU[3]. Upon studying the schematic diagram of the C-64 (Appendix A) the author has determined that there is no other possible origin of the signal except the 6510.

The detailed timing of the 6510 will be discussed in part B of this section.

One addition within the 6510 is the high impedance (tri-state) address buffer. The buffer is enabled by the AEC signal generated by the VIC chip or by the DMA signal which is wired to the expansion port. This feature allows direct RAM/ROM accessing without a conflict with the microprocessor.

A third feature of the 6510 not found in the 6502 is the built in six bit peripheral data register. Several bits of this register control the memory configuration via the PLA. Other bits of the register control several special purpose I/O tasks. The data direction register for this port appears at $0000 and the peripheral data register is at $0001 in the memory map.

Both the 6510 and the 6502 microprocessors are contained in the same size package (40 pin DIP). In order to accommodate the additional peripheral data lines on the 6510, two control signals were eliminated from the 6502 design. These were the set overflow (SO) and SYNC signals. The SO was an input to the 6502 microprocessor used to set the overflow bit of the processor status word. This was not considered, by the author, as a significant loss. The SYNC was an output of the 6502 microprocessor used to signal the fetch of the next instruction opcode. This signal, on the other hand, is extremely important for a single stepping scheme. Because of the lack of a SYNC signal on the 6510, a hardware counter circuit was necessary in order to implement the thesis (see Chapter IV).

The software instruction set is identical in both microprocessors and will be discussed in section 3.

2. Timing and Control

As mentioned above, the 6510 microprocessor functions with two, non-overlapping clock signals which operate between 0 volts and Vcc (+5VDC). The overall Commodore 64 system timing originates as a 14.32MHz. signal referred to as Ø_{color}. This clock signal is utilized by the VIC chip in the video scanning process. Through use of a 74LS193 counter, the signal is reduced to 8.18MHz., referred to as Ø_{dot}. This clock signal is also used by the VIC chip. The VIC chip internally divides the Ø_{dot} clock by a factor of eight which thus produces the 1.02 MHz. Ø_{0in} required by the 6510 microprocessor (known as Ø_1). As mentioned earlier, the 6510 synthesizes the Ø_2 from the Ø_1. The time between each successive Ø_1 defines a machine cycle.

There are three basic machines cycles from which all the 6510 instruction cycles are derived. They are; read cycle - read a byte from the data bus, write cycle - write a byte to the data bus, and an internal cycle which does not affect the bus states. Figure 5 (next page) shows the simplified bus timing for both the read and write cycles.

The following signals comprise the control of the 6510; $\overline{\text{NMI}}$, $\overline{\text{IRQ}}$, R/$\overline{\text{W}}$, $\overline{\text{RDY}}$, $\overline{\text{RESET}}$, $\overline{\text{AEC}}$. The operation of the first five are identical to that of any other typical microprocessor. The $\overline{\text{AEC}}$ as mentioned earlier, isolates the microprocessor connection to the address bus. This feature

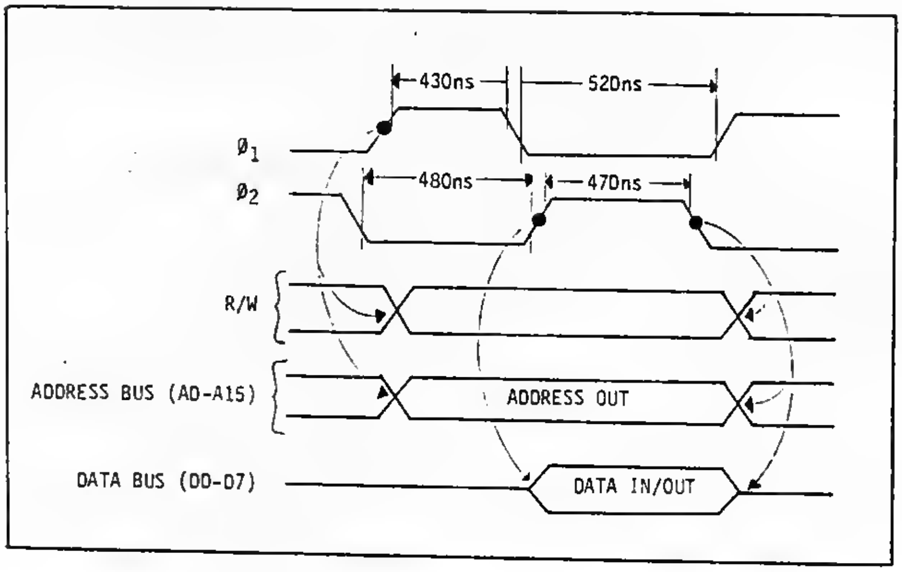


Figure 5 - Read and Write Machine Cycle Timing Diagram

allows other peripheral chips to control the address bus.

The 6510 microprocessor uses the last six memory locations ($FFFA-$FFFF) for the NMI, RESET, and IRQ vectors respectively. The hardware interrupt (IRQ) and the BRK instruction (software interrupt) share the same vector location in memory. It is therefore necessary for the interrupt routine to check the microprocessor status flag during an interrupt to determine whether it was an IRQ or a BRK which had occurred.

The 65xx family of microprocessors, especially the 6510, provide very few external control signals. This tends to complicate the design of the microprocessor system, according to Osborne[4]. For one thing, the 6510 lacks a VMA (Valid Memory Address) signal. It also lacks a DBE (Data Bus Enable). The $Ø_2$ clock must be used in place of the above mentioned signals. Finally, the 6510 does not have an interrupt acknowledge output. Without the acknowledge, the system does not know when an interrupt routine is being executed.

3. Software Instruction Set

The instruction set of the 6510 microprocessor is identical to that of any other 65xx microprocessor. The instructions themselves, are similar to those of all the eight bit machines. The discussion of the software instruction set will be limited to the addressing modes along with the advantages of the "QK02" addressing nomenclature developed and copyrighted by Dr. William H. Pierce (used with his permission). The operation of the stack will also be discussed.

Figure 6 shows the programming model associated with the 6510 microprocessor. As can be seen there is one

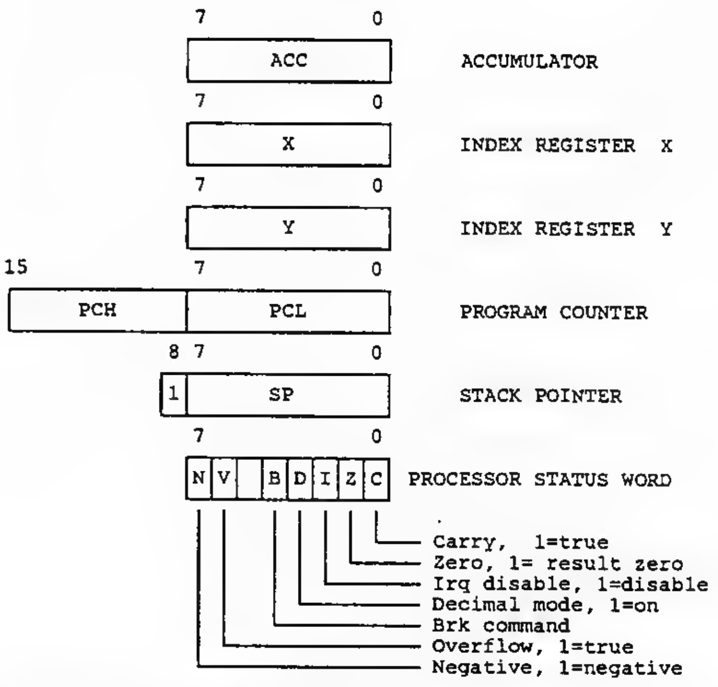


Figure 6 - Programming Model of the 6510 Microprocessor (Reprinted from the C-64 Programmers Reference Manual)

accumulator, two 8-bit index registers, an 8-bit stack pointer, and an 8-bit status register utilizing six status flags.

The X and Y index registers, being only eight bits in length, cannot absolutely index to any memory location (since 64k of memory). Thirteen address modes are provided. These addressing modes give the 6510 microprocessor the versatility that may have otherwise been lost due to the limitations of the index registers.

Table III (next page) shows the entire instruction set of the 6510 microprocessor, it includes the valid addressing modes as well as the extra byte requirements and execution times (in clock cycles). The QK-02 addressing nomenclature was added to the table and will be discussed immediately after the addressing modes are explored.

The thirteen addressing modes, as described in the 6510 literature are as follows:

Immediate	The byte following the instruction is the data byte.
Absolute	The two bytes following the instruction contain the address of the data byte.
Zero Page	The one byte which follows contains the low address of the data. The high address byte is assumed zero (hence zero page, $00hh).
Accumulator	The accumulator is the object of the instruction.

Implied The register is implied by the instruction.

Indirect X (pre-indexed) The address is given in

TABLE III
INSTRUCTION SET OF THE 6510 MICROPROCESSOR
(Reprinted from the C-64 Programmers Reference Guide with modifications by the author)

QK-02 NOMENCLATURE		#			--			-									X)			)Y			X			XX			YY									Y								
INSTRUCTIONS		Immediate			Absolute			Zero Page			Accum			Implied			(Ind.) X			(Ind.) Y			Z. Page, X			Abs. X			Abs. Y			Relative			Indirect			Z. Page, Y			CONDITION CODES					
Mnemonic	Operation	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N	Z	C	I	D	V
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										✓	✓	✓	—	—	✓
AND	A∧M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										✓	✓	—	—	—	—
ASL	C ← [7 0] ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													✓	✓	✓	—	—	—
BCC	BRANCH ON C = 0 (2)																															90	2	2							—	—	—	—	—	—
BCS	BRANCH ON C = 1 (2)																															B0	2	2							—	—	—	—	—	—
BEQ	BRANCH ON Z = 1 (2)																															F0	2	2							—	—	—	—	—	—
BIT	A∧M				2C	4	3	24	3	2																															M_7	✓	—	—	—	M_6
BMI	BRANCH ON N = 1 (2)																															30	2	2							—	—	—	—	—	—
BNE	BRANCH ON Z = 0 (2)																															D0	2	2							—	—	—	—	—	—
BPL	BRANCH ON N = 0 (2)																															10	2	2							—	—	—	—	—	—
BRK	(See Fig. 1)													00	7	1																									—	—	—	1	—	—
BVC	BRANCH ON V = 0 (2)																															50	2	2							—	—	—	—	—	—
BVS	BRANCH ON V = 1 (2)																															70	2	2							—	—	—	—	—	—
CLC	0 → C													18	2	1																									—	—	0	—	—	—
CLD	0 → D													D8	2	1																									—	—	—	—	0	—
CLI	0 → I													58	2	1																									—	—	—	0	—	—
CLV	0 → V													B8	2	1																									—	—	—	—	—	0
CMP	A − M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										✓	✓	✓	—	—	—
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																															✓	✓	✓	—	—	—
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																															✓	✓	✓	—	—	—
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													✓	✓	—	—	—	—
DEX	X − 1 → X													CA	2	1																									✓	✓	—	—	—	—
DEY	Y − 1 → Y													88	2	1																									✓	✓	—	—	—	—
EOR	A⊻M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										✓	✓	—	—	—	—
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													✓	✓	—	—	—	—
INX	X + 1 → X													E8	2	1																									✓	✓	—	—	—	—
INY	Y + 1 → Y													C8	2	1																									✓	✓	—	—	—	—
JMP	JUMP TO NEW LOC.				4C	3	3																												6C	5	3				—	—	—	—	—	—
JSR	(See Fig. 2) JUMP SUB				20	6	3																																		—	—	—	—	—	—
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										✓	✓	—	—	—	—
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	✓	✓	—	—	—	—
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													✓	✓	—	—	—	—
LSR	0 → [7 0] → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0	✓	✓	—	—	—
NOP	NO OPERATION													EA	2	1																									—	—	—	—	—	—
ORA	A∨M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										✓	✓	—	—	—	—
PHA	A → M_S S − 1 → S													48	3	1																									—	—	—	—	—	—
PHP	P → M_S S − 1 → S													08	3	1																									—	—	—	—	—	—
PLA	S + 1 → S M_S → A													68	4	1																									✓	✓	—	—	—	—
PLP	S + 1 → S M_S → P													28	4	1																									(RESTORED)					
ROL	← [7 0] ← [C] ←				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													✓	✓	✓	—	—	—
ROR	→ [C] → [7 0] →				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													✓	✓	✓	—	—	—
RTI	(See Fig. 1) RTRN INT													40	6	1																									(RESTORED)					
RTS	(See Fig. 2) RTRN SUB													60	6	1																									—	—	—	—	—	—
SBC	A − M − $\bar{C}$ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										✓	✓	(3)	—	—	✓
SEC	1 → C													38	2	1																									—	—	1	—	—	—
SED	1 → D													F8	2	1																									—	—	—	—	1	—
SEI	1 → I													78	2	1																									—	—	—	1	—	—
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										—	—	—	—	—	—
STX	X → M				8E	4	3	86	3	2																												96	4	2	—	—	—	—	—	—
STY	Y → M				8C	4	3	84	3	2													94	4	2																—	—	—	—	—	—
TAX	A → X													AA	2	1																									✓	✓	—	—	—	—
TAY	A → Y													A8	2	1																									✓	✓	—	—	—	—
TSX	S → X													BA	2	1																									✓	✓	—	—	—	—
TXA	X → A													8A	2	1																									✓	✓	—	—	—	—
TXS	X → S													9A	2	1																									—	—	—	—	—	—
TYA	Y → A													98	2	1																									✓	✓	—	—	—	—

(1) ADD 1 TO "N" IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE.
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW.
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT.

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
M_S MEMORY PER STACK POINTER

+ ADD
− SUBTRACT
∧ AND
∨ OR
⊻ EXCLUSIVE OR

✓ MODIFIED
— NOT MODIFIED
M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
N NO. CYCLES
# NO. BYTES

zero page by the sum of the X register and the data byte which follows.

Indirect Y (post-indexed) The address is given in zero page by the data byte which follows. The value of the Y-register is then added to this address.

Zero Page X, Zero Page Y The zero page address is given by the byte which follows plus either the X or Y register value.

Absolute X, Absolute Y The address is given by the two byte address which follows plus the X or Y register value.

Relative The address is relative to the opcode location. It is calculated by using the data byte which follows (an 8-bit signed 2's complement).

Indirect The two data bytes which follow contain the location of the address to which the jump will take place.

Using the conventional nomenclature, the desired addressing mode is determined by the number of bytes supplied after the operand, or by cumbersome characters added to the extra bytes. The QK-02 nomenclature alleviates many problems by adding a two character suffix to the end of the mnemonic. Table IV (next page) provides several examples.

As seen from the comparison, the addressing mode can be easily detected with the QK-02 nomenclature (by observing the one or two character suffix to the mnemonic). This is an advantage during a visual inspection of the source code listing. It is also beneficial for the implementation of a source to object code assembler. By examining the suffix (the desired addressing mode), the quantity and value of the extra bytes supplied can be checked for correctness. An assembler of this type was implemented by the author (in Basic programming language), and all prototype programs were written using the QK-02 nomenclature.

TABLE IV

COMPARISON BETWEEN THE CONVENTIONAL AND THE QK-02 NOMENCLATURE

Addressing Mode	Nomenclature			
	Conventional		QK-02	
Immediate	LDY	#$3C	LDY#	$3C
Absolute	LDA	$304B	LDA--	$304B
Zero Page	LDX	$80	LDX-	$80
Accumulator	ROL		ROL	
Implied	TAX		TAX	
Indirect X	LDA	($1C,X)	LDAX)	$1C
Indirect Y	LDA	($1D),Y	LDY)Y	$1D
Zero Page X	LDA	$15,X	LDAX	$15
Zero Page Y	LDA	$FF,Y	LDAY	$FF
Absolute X	STA	$3C74,X	STAXX	$3C74
Absolute Y	STA	$B47C,Y	STAYY	$B47C
Relative	BEQ	$C0	BEQ	$C0
Indirect	JMP	($F030)	JMP()	$FO30

The stack pointer of the 6510 is eight bits in length, thus providing a maximum of 256 continuous bytes of stack memory. The stack is non-relocatable. Through its internal hardware, the stack is permanently mapped into page one of memory ($0100 - $01FF). An interesting aspect about the stack worth mentioning is that the stack is circular in operation. If the stack is incremented past $01FF, it recycles to $0100. This is true of the converse also, if it is decremented below $0100, it recycles to $01FF. No error is generated during this process. This feature is not readily useful as a programming tool. In fact, it can be detrimental if many stack manipulations are present in the target program.

IV PROTOTYPE HARDWARE

The purpose of the hardware portion of the project is to cause the 6510 microprocessor to single step from instruction to instruction. Due to the design of the Commodore 64 and the 6510 CPU, single stepping cannot be achieved without this external hardware. By the definition of the interrupt process; if an interrupt occurs during an instruction, the instruction is carried out entirely and the microprocessor control is then passed to the interrupt routine. The interrupt is therefore the ideal way to generate single stepping in the 6510 microprocessor. Since the 6510 is not equipped with a SYNC output, the interrupt (non-maskable interrupt, NMI) must be of the "timed interrupt" type as mentioned in Chapter II.

The hardware portion can be broken down into two major phases, the design and the implementation. This chapter will encompass these aspects of the thesis project.

A. Prototype Hardware Design

To insure a viable design, the design criteria for the interrupt circuit needs to be established. This will be discussed in the following section. The circuit operation and the hardware/software timing will then be analyzed later in this chapter.

1. Design Criteria

In order to create the necessary timing circuit, a set of guidelines were developed. These guidelines were established early in the prototype hardware design stage. The basic design criteria is as follows:

a) The circuit must count a predetermined number of $\emptyset_2$ signals before generating an interrupt.

b) The predetermined value mentioned in (a) must be user selectable (at least during initial implementation).

c) A pulse from the C-64 via the Pb_0 of CIA2 must:

   1) Cause the circuit to load the count value.

   2) Start the counting of $\emptyset_2$ pulses.

d) When the counter reaches zero, a high-to-low-to-high level pulse must be generated. The pulse, in turn, must:

   1) Generate a NMI within the C-64.

   2) Halt the counting of $\emptyset_2$ pulses.

e) The circuit must latch into its inactive state after power up.

f) The circuit must meet the power and frequency requirements of the Commodore 64 computer:

   1) $I < 50$ma @ 5VDC.

   2) $f = 1$MHz.

g) Since the interrupt output will be hardwired into the 6510's NMI, a disconnecting means must be provided to free up the NMI for the keyboard generated NMI.

A timing circuit was designed and implemented to meet the above mentioned criteria.

2. Circuit Operation

The basic operation of the circuit is fairly straightforward. A pulse from the Commodore 64 causes a counter to load a predetermined value. It then starts decrementing the value, with each $\emptyset_2$ pulse, until zero is reached. Upon reaching zero a NMI is generated in the Commodore 64.

Figure 7 shows the functional diagram of the counter circuit. All integrated circuits were chosen to be of the low power Schottky design in order to minimize the current

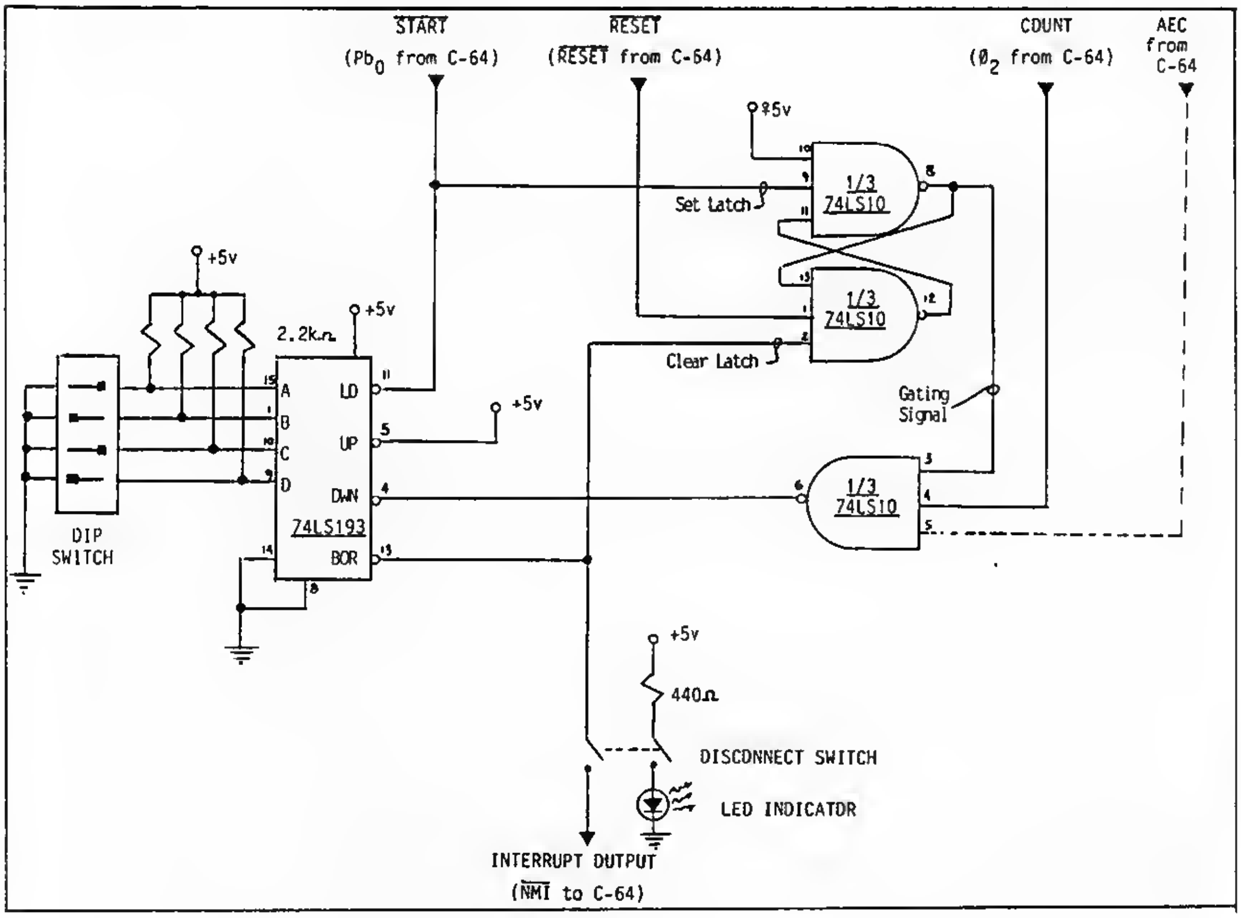


Figure 7 - Functional Diagram of the NMI Timer

drain on the C-64. The counter IC chosen was a 74LS193. It was utilized because of its count down capabilities, borrow output, and its presetability. The 74LS10 (triple three input NAND) has a twofold purpose. First of all it generates the count down pulse used by the 74LS193. This pulse is synthesized by the logical NANDing of the 6510's $\emptyset_2$ signal and the gating signal. The AEC input to this NAND gate was eliminated during the implementation stage. This modification will be discussed in part B of this chapter. The second use of the 74LS10 is to comprise a latch which generates a gating signal which, in turn, controls the counting process. The detailed circuit operation will follow.

The START pulse (Pb_0 of the C-64) causes the 74LS193 to load a preset, 4-bit, binary value. This counter value is user selectable through a DIP switch. The START pulse also sets a latch which generates the gating signal. With the gating signal active (high), the NAND gate will start passing the $\emptyset_2$ signal through to its output, inverted of course. This signal is then called the DWN pulse which is applied to the 74LS193. With each DWN pulse, the counter value will be decremented by one. Upon reaching zero, the BORROW output of the 74LS193 becomes active (low) which clears the latch (halting count DWN pulses), and through a hardwired connection, generates a NMI within the Commodore 64. Since the NMI connection is hardwired, a disconnecting switch is provided so that the circuit's NMI connection can be isolated from the computer. It should be noted that the circuit output (NMI) is standard TTL logic rather than open-collector. This will be discussed later in this chapter. An LED is also provided which indicates the logic of the isolation switch. The Commodore 64 RESET signal is connected to the latch as an additional clear input. This ensures that the latch will be cleared and no counting will take place upon power up of the computer.

Data sheets on all the integrated circuits utilized in the timer as well as pertinent ICs in the Commodore 64 itself can be found in Appendix A.

3. Timing/Software Interaction

The details of the software and the associated timing of the single stepping circuit will be discussed in this section. The overall prototype software design and implementation will be presented in chapter V.

The following scenario gives the sequence of events preceding the execution of one target program instruction and the NMI which occurs during it.

a) Push the previous processor status word (PSW) and the address of the next target program instruction on the stack.

b) Restore all microprocessor registers, except the X register (X-register chosen arbitrarily), to their previous value, that is, the value during the last target program instruction.

c) Use the X register to send the output pulse, via the CIA, to the timer circuit. The timer will now start counting 0_2 pulses.

d) Restore the X register to its previous value.

e) Perform a RTI which will reload the PSW and the address of the next target instruction from the stack.

f) The target program instruction will now be executed and an NMI will be generated during the instruction.

As seen from the above scenario, the timer will be active (counting $\emptyset_2$ pulses) from events c-f. This represents the time total delay which must be provided before the interrupt can take place. Table V is a

TABLE V
REALIZATION OF DELAY TIME FOR A TIMED NMI

Instruction	Purpose	# cycles
LDA-- $0345	Get high byte of next inst.	(a)
PHA	Push on stack.	(a)
LDA-- $0349	Get low byte of next inst.	(a)
PHA	Push on stack.	(a)
LDA-- $0343	Get previous PSW.	(a)
PHA	Push on stack.	(a)
LDY-- $0340	Restore previous Y reg. value.	(a)
LDA-- $0342	Restore previous Acc. value.	(a)
LDX# $00	Set X reg to $00.	(a)
STX-- $DD01	Send low pulse to timer circuit via the CIA.	(a)
INX	Increment X reg. to $01.	(a)
STX-- $DD01	Send high pulse to timer circuit via the CIA.	(b)
LDX-- $0341	Restore previous X reg. value.	4
RTI	Return from interrupt...	6
.		
XXX	Target instruction.	
	Total delay (in cycles)	10

Notes:
(a) Instruction length is not critical since timer has not started.
(b) Timer will start counting at the end of this instruction.

realization of the time delay using the actual instruction sequence implemented by the author in the prototype system. Addresses $0340-$0346 contain the stored register values from the last target instruction executed. This is explained with the programming memory map in Chapter V (Table IX, page 58).

Table V indicates that a total of 10 $\emptyset_2$ cycles should pass before an interrupt is generated. This theoretical value does not take into consideration any propagation delay which may be present in either the CIA (in the C-64) or the 74LS193.

Figure 8 (next page) shows the timing diagram associated with the software/hardware. Note that the timing diagrams for the 74LS193 and the CIA (6526) have been provided in Appendix A. There are several points worth mentioning with regard to the timing. First of all, the LOAD input to the 74LS193 is asynchronous in nature. Upon the falling edge of this input the preset value is loaded. Decrementing of the preset counter value is not possible until the LOAD input is brought back to its inactive (high) state. Secondly, the output of the 74LS193 (timer value) is decremented on the rising edge of the count DWN pulse. Once enabled, the count DWN pulse is in effect the inverted $\emptyset_2$ clock signal.

The propagation delay through the NAND gate and counter is approximately 20-50 nanoseconds. This delay is considerably smaller than the 1 microsecond $\emptyset_2$ clock, it can therefore be neglected. On the other hand, there is a

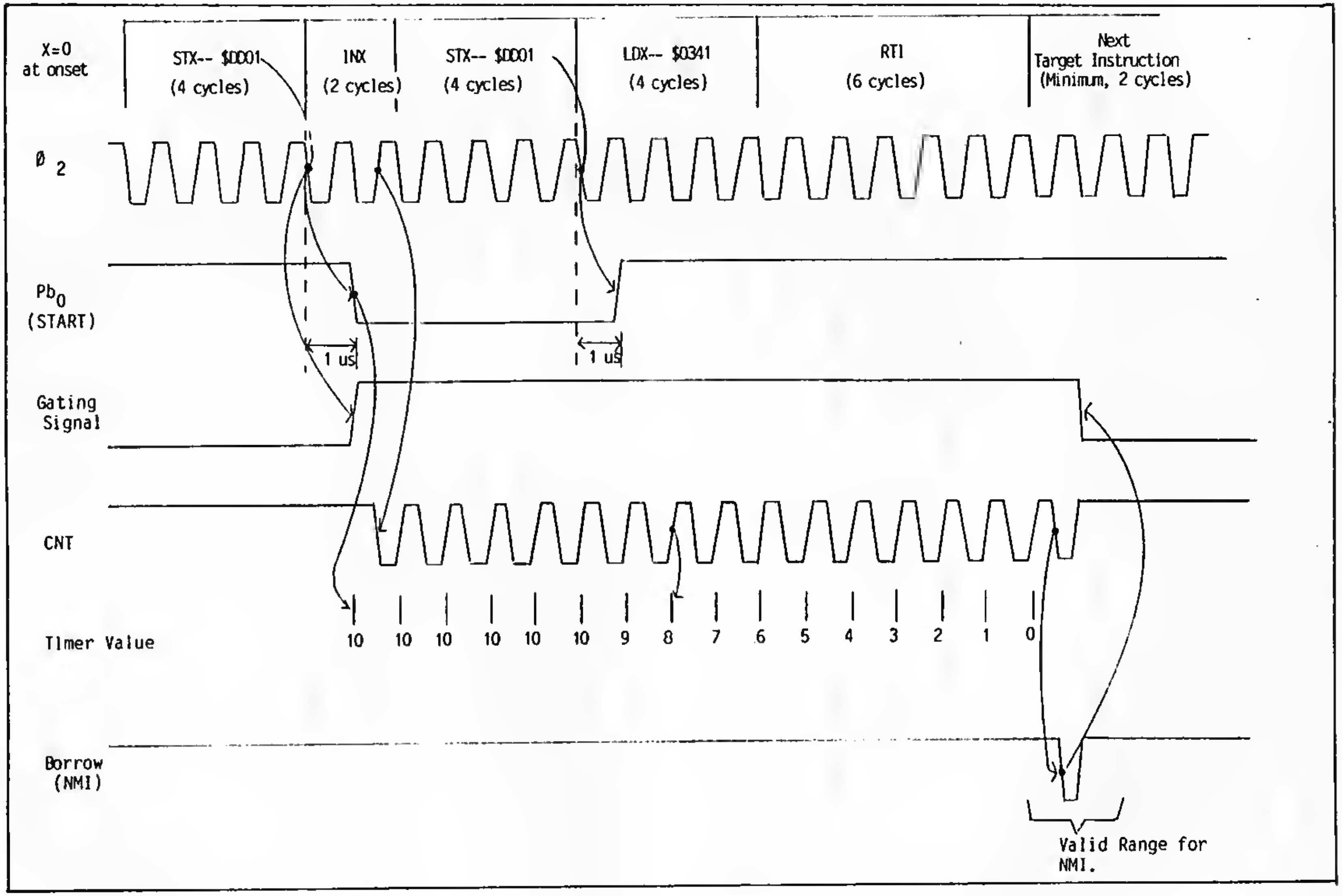


Figure 8 - Timing Diagram of the Hardware/Software Interaction

significant delay incurred when data passes through the CIA in the Commodore 64. This is due to the fact that data is not valid on the peripheral data register of the CIA until the next falling edge of the $\emptyset_2$ clock. This delay, one additional clock cycle, must be included when determining the overall counter value. The delay is indicated on figure 8.

B. Prototype Hardware Implementation

Once the circuit and associated software is designed, it must be implemented in a reasonable manner. This section will address the physical attributes of the hardware as well as an analysis of the final counter value.

1. Auxiliary Port

In order to implement the circuit several control and timing signals were needed externally to the Commodore 64. These were the $\emptyset_2$, Pb_0 (from CIA2), $\overline{RESET}$, $\overline{NMI}$, 5v, Gnd, and $\overline{AEC}$. Referring to Table II (page 22) it can be seen that not all of the above mentioned signals are available on any one port. Rather than using both the User Port and the Expansion Port (and still not being able to facilitate a connection to $\overline{AEC}$), an auxiliary port was installed in the author's Commodore 64.

The auxiliary port is a 15 pin, miniature type port. It is mounted directly to the main printed circuit board of the Commodore 64. Due to the design of the Commodore 64 enclosure, the rear exposure of the computer was easily modified to provide access to the port. The internal connections from various points on the PC board to the port itself are made with 24 AWG, solid wire.

2. Circuit Construction

The timing circuit is mounted on a 2" x 4", single sided, printed circuit board. All the discrete components are soldered directly to the PC board. The two integrated circuits are mounted via IC sockets. Several unused pins of the IC sockets have been removed. This allows additional room for the placement of the traces on the PC board.

Since the $\overline{\text{RESET}}$ control was already utilized in the prototype circuit, it was later decided to provide a momentary contact push-button to cause a reset sequence in the computer. This was extremely useful during prototype software debugging and implementation.

Figure 9 shows the circuit board layout. Mounted on the circuit board is the matching connector for the auxiliary port located on the Commodore 64.

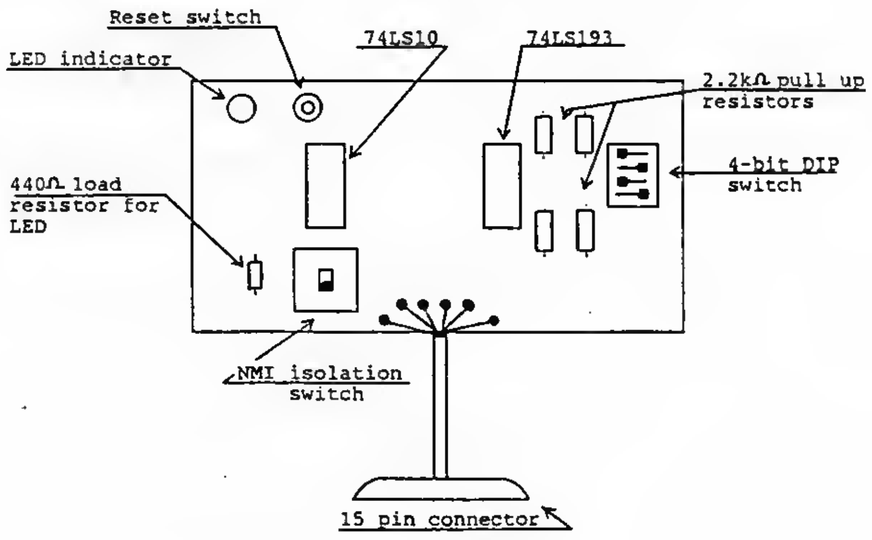


Figure 9 - Layout of the Prototype Timing Circuit Board

The output of the timer circuit (NMI) is not open-collector logic as it should be. The normal TTL output was provided because, orginally, the output was to be connected to a single shot multi-vibrator in the Commodore 64. It was later realized that the multi-vibrator would have generated a NMI with too long of a duration. This would require a software delay loop with the NMI routine, thus slowing down the single stepping. The circuit was alreacd prototyped when this design change occured. The author realizes that connecting standard TTL to a line with multiple open-collector output, such as a NMI input to a microprocessor, is not a good design practice. The isolation switch (see figure 7) allows the disconnection of the NMI when the circuit is not in operation.

3. Counter Value Implementation

It was originally thought the AEC input to the 6510 microprocessor was utilized in its timing sequence. Because of this it was included as an additional input to the NAND gate (along with $\emptyset_2$) which generates the count pulses. During the implementation stage it was determined that this was an incorrect assumption. The signal is currently disconnected on the prototype circuit.

The Video Interface Chip (VIC) controls the AEC and uses it to cause the address buffer within the 6510 to enter a high impedance state (see Chapter III). The Bus Available (BA) signal from the VIC is connected to the Ready (RDY) input on the microprocessor. Using the above mentioned scheme, the VIC chip can "steal" $\emptyset_2$ cycles from the 6510.

Since this is the case, it seems that this should disrupt the counter value. Experimental results show that the counter works properly in all cases. One explanation may be that one "stolen" cycle (ie. the instruction currently being executed will take one cycle longer to execute) still allows the NMI to fall upon the correct target instruction. Not enough documentation was available for the 6510 CPU and the VIC for the author to draw a concrete conclusion regarding this discrepency.

Although the calculated counter value of ten was indeed correct, at the time of the original circuit design this value was not known with much confidence. The DIP switch was utilized so that the value could easily be adjusted if necessary.

Having the counter value adjustable lead to experimentation with regard to other counter value and results. Table VI summarizes these results.

TABLE VI
SUMMARY OF INTERRUPT TIMER VALUES

Timer Value	Result
0	Interrupt does not occur.
1-2	Interrupt occurs during RTI instruction, stuck in loop.
3-8	Interrupt occurs during proper target instruction, but registers are not updated.
9	Works in all cases.
10	Works in all cases.
11-14	Works for some longer instructions only.
15	NMI misses the target instruction completely.

The table data helps support the VIC "cycle stealing" reasoning mentioned earlier, since both nine and 10 counts cause a proper interrupt.

V. PROJECT SOFTWARE

The software portion of this thesis research involves the creation of various command routines and support subroutines which together form the debug monitor program. Since this thesis was not intended as an exercise in assembly language programming, only major points will be addressed. These points will include the software design criteria, the overall program logic, single stepping logic, break point logic, and the general implementation scheme.

Appendix B contains the complete program listings. The organization within the individual source file listings is discussed in section B of this chapter.

Appendix C provides several sample outputs, via screen dumps, along with a detailed explanation of what command sequence took place.

A. Prototype Software Design

As with the hardware, the software begins as a set of design criteria. Using this criteria, the general program logic is formed.

1. Design Criteria

The software design criteria involve definite facts, but it also includes several design assumptions. These assumptions are made in order to anticipate the user. Care was also taken to provide a system which was ergonomically comfortable.

The basic design criteria is:

a) Implement the "Ideal" features indicated in Chapter II.

b) Utilize as many Kernal (operating system) routines as possible to minimize programming.

c) Utilize a jump table so that as the routines are moved in memory, program changes are minimized. The object code will therefore not be relocatable except by rewriting the jump table.

d) The final software product will occupy a maximum of 4K bytes of RAM memory residing from $C000 to $D000.

e) During single stepping, one line will be output to the screen per instruction.

f) It is assumed that the target program will always be in RAM memory anywhere from $2000 to $A000.

g) It is assumed that during single stepping, if a screen display is not selected, the target code is free of invalid opcodes.

Most of the above mentioned criteria were followed when designing the necessary software. Due to assembler and memory limitations, the final programs, as presented in this manuscript, do not reside in memory from $C000-D000 (item d). Instead they are sparsely located (to allow for modifications during the programming stage) in memory from $8400-C000. The program segments would collectively fit, however, into the 4K range set forth in item (d) if the assembler and memory had allowed it.

2. Program Logic

The general program logic of the debug monitor is shown in figure 10.

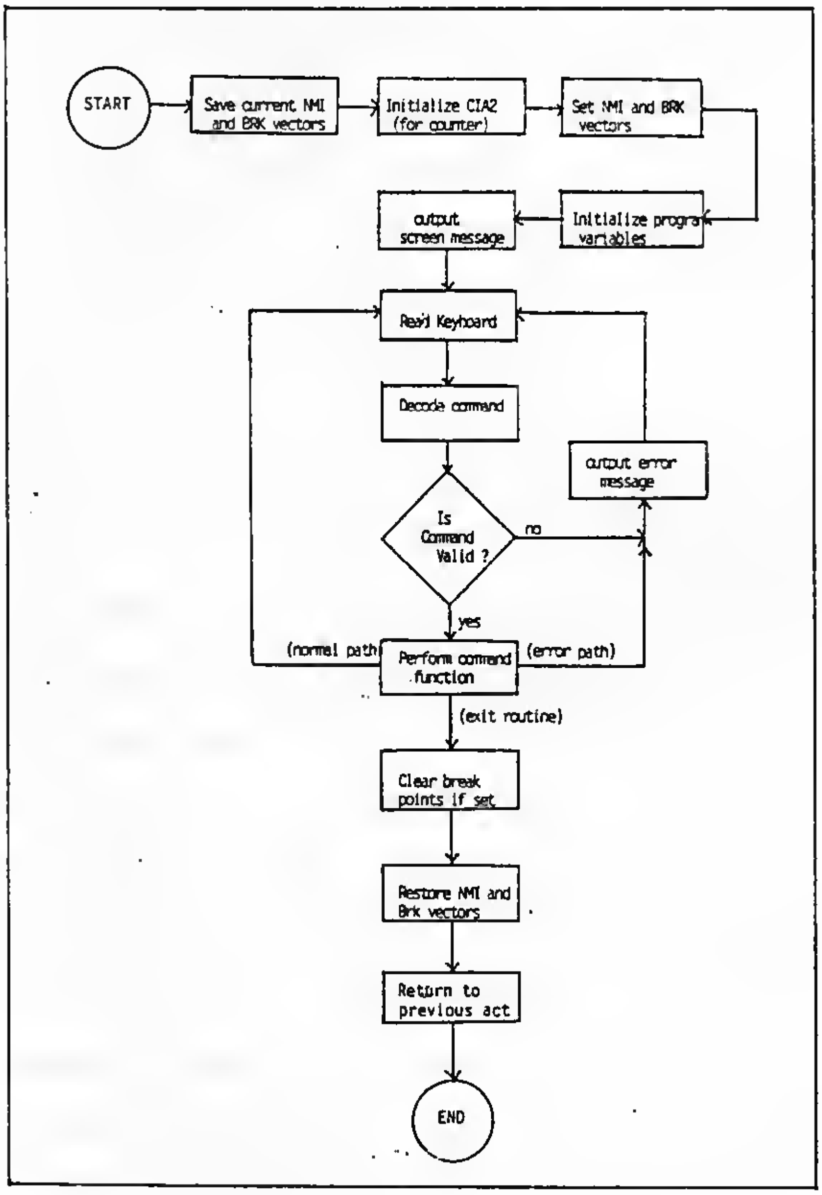


Figure 10 - Flow Chart of Debug Monitor

The logic begins by initializing the program parameters. This includes configuring the CIA for an output port to pass the START pulse to the counter circuit. From here, a loop - the debug monitor itself, is entered. All commands are issued from within this loop. These debug commands are generalized in the box entitled "Perform command function". Table VII (page 55) summarizes the commands implemented by the author. Upon issuance of the "Exit" command, the monitor loop is exited, parameters are reset, and the program terminates.

The logic for single stepping is considerably more complex. Discussed here is the portion of the single stepping routine which called upon via the NMI vector. The segment executed preceding the NMI (register restore and generation of the counter start pulse) has been presented in Chapter IV with the prototype hardware. Figure 11 (next page) shows the flow chart of the single step routine.

Upon NMI, the microprocessor registers must be stored in program memory RAM (see Table IX, page 58). Unlike most CPUs, when an interrupt occurs on the 6510, only the program counter and processor status word are stored on the stack. There is a NMI software routine first called upon in the Kernal (operation system) which subsequently pushes the microprocessor's registers onto the stack. The author's program copies the contents of the stack to the program's storage registers. After storing the register data, one line of display is written to the screen. This display consists of the target instruction's address,

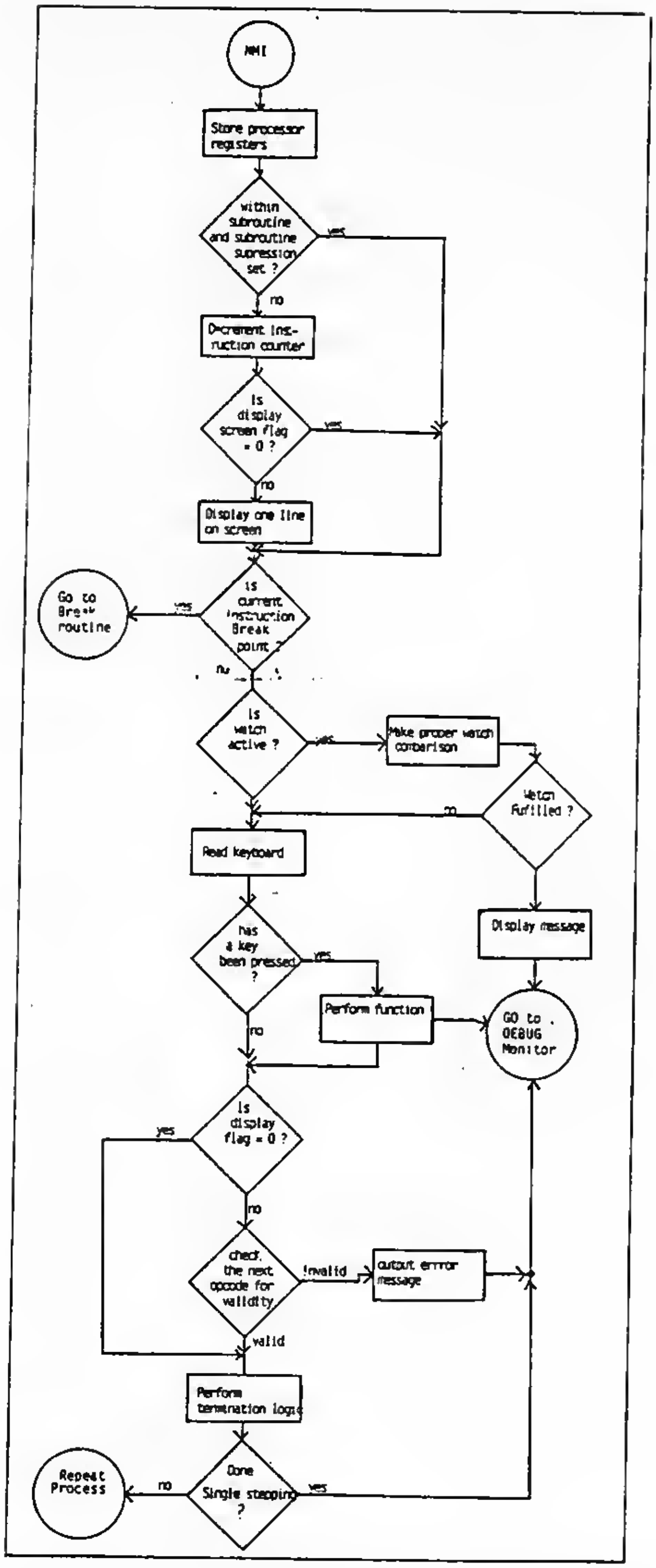

NMI
Store processor registers
within subroutine and subroutine supression set ?
yes
no
Decrement instruction counter
Is display screen flag = 0 ?
yes
no
Display one line on screen
Go to Break routine
yes
Is current instruction Break point ?
no
Is watch active ?
yes
Make proper watch comparison
Watch Fulfilled ?
no
Read keyboard
Display message
has a key been pressed ?
yes
Perform function
GO to DEBUG Monitor
no
Is display flag = 0 ?
yes
no
check the next opcode for validity
invalid
output error message
valid
Perform termination logic
Repeat Process
no
Done Single stepping ?
yes

Figure 11 - Flow Chart of Single Step Routine

disassembled object code (in QK-02 nomenclature), operands, and the microprocessor registers. The processor status word is displayed in binary so that the individual flags can be easily discerned. A time delay loop is executed after the screen display to allow the user time to analyze the screen information. This time delay is user adjustable. If a screen display is not selected or if the subroutine suppression parameter (executing through an entire subroutine as one single step) is active, the display and delay portion of the program is bypassed. Next the current opcode is checked to see if whether it is the BRK (software interrupt) opcode. If so, control is passed to the break point routine which will be discussed later. Following this, the watch parameter flag is tested. If active, the proper watch comparison (such as comparing the X-register with some value) is performed. Upon a match, control is passed to the debug monitor. Next, if the user has selected a screen display, the validity of the next target program instruction is checked. This is done to prevent the 6510 CPU from "locking up" when it attempts to execute a invalid or undefined opcode*. The opcode validity is checked by searching the disassembly table. The search routine is relatively slow but negligible when compared to the screen display delay. The opcode validity check only occurs when a screen display is utilized. This greatly speeds the

*Other CPUs, especially the 16-bit machines, have built in error handling vectors to prevent this "lock-up" from occurring.

execution in the non-screen display modes. Finally, if more single stepping is required, the registers are restored and the counter circuit generates another NMI. Otherwise, control is passed back to the debug monitor.

The break point logic is fairly simple. It is presented in figure 12. One should keep in mind that the operation of the two break points is identical.

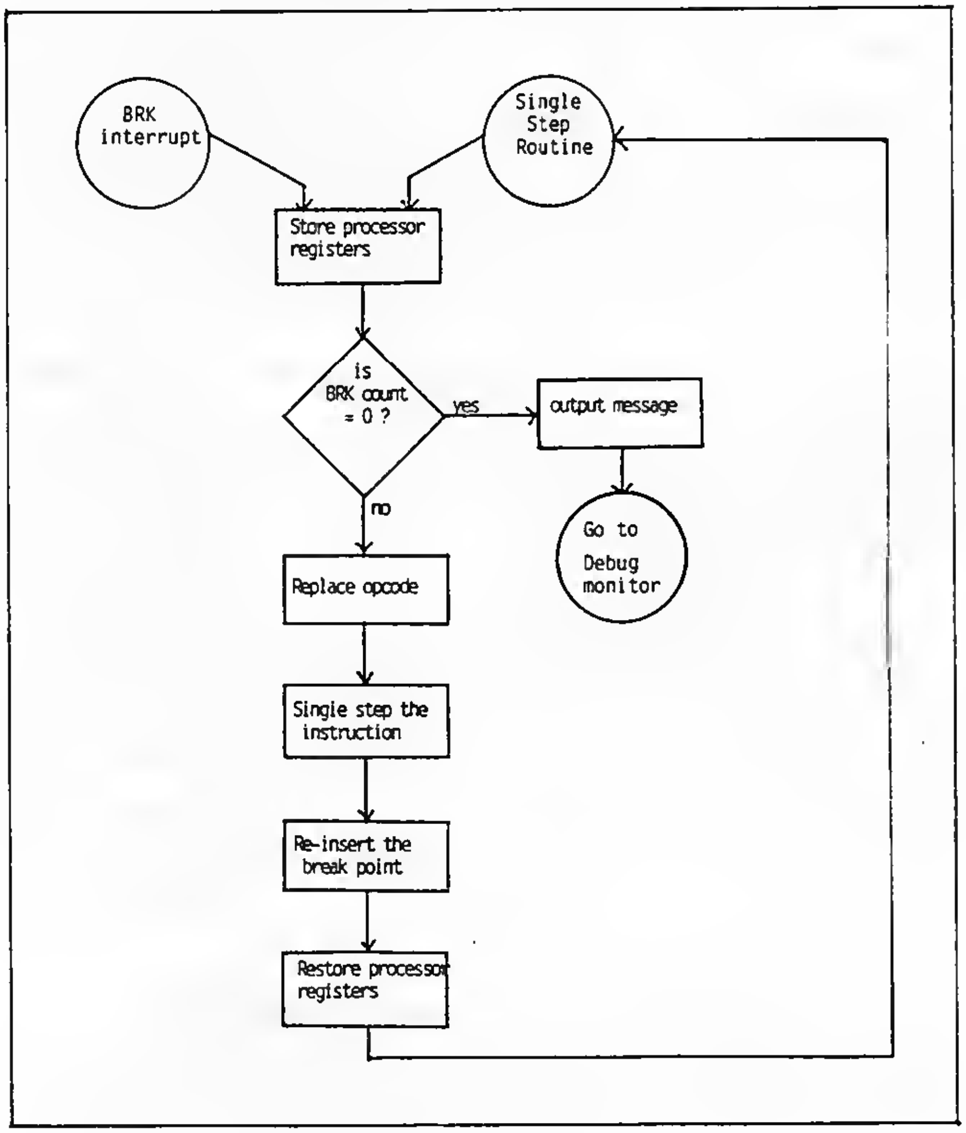

BRK interrupt
Single Step Routine
Store processor registers
is BRK count = 0 ?
yes
output message
Go to Debug monitor
no
Replace opcode
Single step the instruction
Re-insert the break point
Restore processor registers

Figure 12 - Flow Chart of Break Point Logic

As seen from the figure, the break routine can be entered by two means. First of all, it can be called upon via the software interrupt vector when a BRK occurs during full speed program execution (not single stepping). Secondly the routine can be entered from the single step routine itself. During single stepping, the interrupt mask bit is set in the microprocessor. Because of this, the software interrupt generated by the BRK instruction is masked. Therefore, a comparison (is opcode = BRK?) must made in the single stepping software routine as mentioned earlier. The remainder of the break routine checks if the break point is to be recognized or bypassed until the n-th detection of it. If it is to be bypassed, this is accomplished by re-writting the proper opcode under the break point and single stepping the instruction. The break point is then replaced until its next execution when the process repeats. Upon the n-th detection (BRK count should equal 0) control is passed to the debug monitor.

There is one important note about break points. Since the target code is overwritten by the BRK instruction, if an exit command is issued, any current break points must be reset. This ensures that the target code is left intact after the debugger has acted upon it.

B. Prototype Software Implementation

Table VII (next page) contains a summary of the final commands implemented by the author. The commands are shown in their proper syntax. An important differentiation

TABLE VII
SUMMARY OF DEBUG COMMANDS IMPLEMENTED

General Operational Commands:

X	Exit and restore system.
D hhhh	Disassemble 20 instructions in the QK-02 Addressing Nomenclature.
M hhhh [jjjj]	Display memory range $hhhh to $jjjj in both hexidecimal and ASCII. {space bar} - freeze display 'Q' - abort command.
F hhhh [jjjj] xx	Fill memory range $hhhh to $jjjj with value $xx.
G hhhh	Jump to location $hhhh and begin execution, completely uncontrolled.
L "name" [jjjj]	Load file from disk, if $jjjj is specified, the file will be loaded at that address, otherwise it is loaded back to the address from which it was saved.
S "name" hhhh jjjj	Save memory range $hhhh to $jjjj on disk.

Trace/Execute setup commands:

P	Display program paramters (If and where break points are set at, if subroutine supression is active, if and to what value a watch is set at.
B1 hhhh	Define break point #1 at $hhhh.
B2 hhhh	Define break point #2 at $hhhh.
CB	Clear either or both break points.
DR	Display microprocessor registers.
SS	Set subroutine supression parameter.
CS	Clear subroutine supression parameter.
LX hh	Load X register with the value $hh.
LY hh	Load Y " " " " "
LA hh	Load Accumulator " " " "
LP hhhh	Load program counter with $hhhh.
WX hh	Watch the X reg for $hh.
WY hh	Watch " Y " " "
WA hh	Watch " Acc. " "
WM hhhh jj	Watch memory address $hhhh for $jj.
WW hhhh jjjj	Watch address word (two consecutive bytes) starting at $hhhh for $jjjj.
CW	Clear watch parameter.

Trace/Execute Single Step Commands:

E hh	Execute (single step <u>without</u> screen display) $hh instructions and halt.
T hh	Trace (single step <u>with</u> screen display) $hh instructions and halt.
EJ	Execute until Jump opcode is encountered.
TJ	Trace " " " " "
EC	Execute continously.
TC	Trace "
EB hh	Execute until break point is encountered $hh times and then break at it.
TB hh	Trace until break point is encountered $hh times and then break at it.
HS	Hide Subroutine - execute the remainder of the subroutine (single step without screen display).

During all Trace/Execute commands, the following keys are scanned:

{csr up}	- increase trace speed (Trace mode only).
{csr dn}	- decrease " " " " "
S	- Single step to next instruction.
Q	- Abort command
{space}	- freeze display

Notes:

xx, hh	represents 8-bit hex values.
hhhh	represemts 16-bit hex values.
[jjjj]	optional input.

should be made between "trace" and "execute" pertaining to single stepping. Through the programs, trace indicates single stepping through the object code with a screen display. Execute, on the other hand, refers to the single step execution of the object code without the screen display.

It it beyond the scope of this manuscript to attempt to explain the programming of each command sequence. The

TABLE VIII
SUMMARY OF PROGRAM SEGMENTS IN SOURCE FILE LISTINGS

Source File		Contents
DRIV.SO		Program initialization, main debug loop.
SUB1.SO	subroutines	Binary to ASCII hex output, Binary to ASCII binary output, clear input buffer, read keyboard, error check, set system vectors.
SUB2.SO	subroutines	Decode keyboard, add value to zero page indexes, compare zero page indexes.
SUB3.SO	subroutines	Output registers.
CMD1.SO	commands	Display memory, Hide subroutine, Fill memory.
CMD2.SO	commands	Disassembly,
CMD3.SO	commands	Load memory, Save memory, Display registers, Exit, Set break.
CMD4.SO	commands	Clear, show parameters.
CMD5.SO	commands	Go, Load registers, Watch registers, Subroutine supression, Trace/Execute setup, NMI setup.
CMD6.SO	commands	NMI single step routine.
MESS.SO		ASCII messages and reference table.
BRK1.SO		BREAK interrupt routine.
JUMP.TBL		JUMP table.

various commands are organized in ASCII files (see Appendix B). Several commands and/or subroutines are consolidated in each ASCII file. Table VIII provides a summary and breakdown of the contents of each file. The analysis of the programming for each command sequence is left up to the reader, using the source code of Appendix B.

The assembler utilized by the author could not assemble the approximately 1,000 lines of source code if contained in one source file. Therefore, the source code has been seperated into program segments, as mentioned earlier. By utilizing a jump table, as routines (or program segments) were moved or expanded, only the reference in the table required changing. This is a great advantage over direct calls to routines since, in the later case, all references to the altered routine would need to be found and modified. The additional jump imposed on each routine call offers a negligible execution speed decrease. A jump table was also utilized in the Commodore 64's kernal (operation system). This allowed easy software interfacing with the operation system.

Table IX (next page) provides a memory map of the various memory locations used for registers, flags, and counters. It would have been advantageous to utilize zero page locations for these program variables (from a program size standpoint - zero page addressing requires one less operand byte). The majority of zero page is controlled by the kernal. The author felt the integrity of this data might be in jeopardy since an unforeseen kernal routine could disrupt the data. The data is located in the range from $033C to $03FF. This area is reserved for cassette tape I/O buffering. Since a cassette tape drive is not present on the author's system, the data should be safe.

TABLE IX
MEMORY MAP OF PROGRAM REGISTERS, FLAGS, AND COUNTERS

LOCATION	DESCRIPTION	CONTENT/LOGIC
033C	ERROR FLAG	$00=OK, ELSE CONTAINS ERROR #.
033D	SUBROUTINE SUPPRESSION FLAG	$00=OFF
033E	WATCH FLAG	INDICATES IS WATCH IS ACTIVE $00=OFF $01=WATCH Y $02=WATCH X $03=WATCH ACC $04=WATCH MEM $05=WATCH 2 BYTE WORD
033F	WATCH VALUE	VALUE TO WATCH FOR (EXCEPT WORD)
0340	Y REGISTER	CONTENTS OF REGISTER BEFORE NMI
0341	X REGISTER	" " " " "
0342	ACCUMULATOR	" " " " "
0343	PROCESSOR STATUS REGISTER	" " " " "
0344	LOW ADDR. OF CURRENT OPCODE	" " " " "
0345	HIGH ADDR. OF CURRENT OPCODE	" " " " "
0346	STACK POINTER	" " " " "
0347	LOW ADDR. OF NEXT OPCODE	" " " " "
0348	HIGH ADDR. OF NEXT OPCODE	" " " " "
0349	MEMORY WATCH LOCATION - LOW	LOW ADDR OF MEMORY WORD WATCH
034A	MEMORY WATCH LOCATION - HIGH	HIGH ADDR OF MEMORY WORD WATCH
034B	TRACE/EXECUTE FLAG	$00=NO DISPLAY (EXECUTE) $01=DISPALY (TRACE)
034C	LOW ADD OF 8RK PT #1	
034D	" " " " " .#2	
034E	HIGH ADD OF BRK PT #1	
034F	" " " " " #2	
0350	OPCODE UNDER BRK PT #1	
0351	" " " " #2	
0352	BREAK COUNTER	# OF BREAKS BEFORR STOPPING.
0353	8REAK POINT OFFSET	INDICATED WHETHER BRK POINT #1 OR #2 HAS BEEN HIT.
0354	SCREEN DISPLAY INTERVAL TIMER	$80, INITIAL VALUE
0355	INSTRUCTION COUNTER	# OF INSTRUCTIONS TO BE EXECUTED OR TRACED BEFORE STOPPING.
0356	TRACE CONTROL FLAG	CONTAINS 2ND CHAR OF 'TRACE' OR EXECUTE CMD
0357	SUBROUTINE LEVEL COUNTER	USED WHEN SU8R SUPP IS ACTIVE TO INDICATE THE LEVEL OF SUBR.
0358	BREAK JUST HIT FLAG	INDICATES THAT A BRK PT WAS HIT AND THE ORIGINAL OPCODE NEEDS TO BE WRITTEN OVER AS '00'.
0359	HIDE SUBROUTINE FLAG	$01= HIDE SUBROUTINE ACTIVE.
035A	WATCH LOW	LOW BYTE OF WORD TO WATCH FOR
035B	WATCH HIGH	HIGH BYTE OF WORD TO WATCH FOR
035C	NOT USED	
035D	BREAK VECTOR LOW	CONTENTS OF $0316 AT PGM START
035E	BREAK VECTOR HIGH	" " $0317 " " "
035F	NMI VECTOR LOW	" " $0318 " " "
0360	NMI VECTOR HIGH	" " $0319 " " "

CONCLUSIONS AND RECOMMENDATIONS

The major conclusion which can be drawn from the thesis research is that the various microprocessors available have different degrees pertaining to if, and how easily, they can be utilized for debugging purposes. The operation of single stepping was fully investigated.

Secondly, once a microprocessor (or for that matter an entire computer system) is chosen, the debugging system should be customized around the CPU's features. This design style will take advantage of the microprocessors high points while making up for the machines deficiencies.

Specific to the Commodore 64, it was concluded that the user lacks much information with respect to the 6510 microprocessor and the Vidio Interface Chip's operation. This could be attributed to the fact that the Commodore 64 contains proprietary integrated circuits whose design is owned by Commodore Buisness Machines Inc.

It was concluded, as expected, that the execution speed of the source code was reduced significantly. Table X (next page) provides a summary of this information.

Several recommendations can be made, most of which pertain to the hardware.

First of all, the standard TTL output of the counter circuit should be buffered with an open-collector driver such as a 74LS07 buffer/driver or a NAND gate utilized with

TABLE X
RELATIVE EXECUTION SPEED
(BASED ON THE EXECUTION OF 12,288 NOP INSTRUCTIONS)

		Time Increase
Actual execution time	25.08 MSec.	-
Execution mode	3.9 Sec.	156x
Trace mode (full speed)	878 Sec.	34,996x
Trace mode (min. speed)	5342 Sec.	213,022x

open-collector output. This will provide a proper connection to the 6510's NMI input.

Secondly, the author had removed several of the unconnected pins of the IC sockets. This eliminated the need for wire jumpers on the single-sided circuit board. One problem which developed was that the IC themselves would loosen in the sockets. It should be recommended that the unconnected pin removal not be done on future prototypes. Either jumpers can be installed on future prototypes or double-sided printed circuit boards can be utilized.

Since the AEC was not needed in the final design, the auxiliary port installed on the Commodore 64 was not absolutely necessary. The choice can be left up to the reader as to whether he/she should utilize both the Expansion and User ports to obtain the necessary signals, or modify the computer with the auxiliary port as the author did.

Regarding the software, when the disassembly routine was written, a table containing mnemonic and operand data was required in memory. The table was specifically designed for disassembly. If assembly were required, a different table structure would be desirable. A recommendation should be made to modify the disassembly routine to scan a new table which could be utilized for both assembly and disassembly. This would greatly reduce memory usage when assembly is implemented.

One additional command could be of great use in a debugging utility. This is a variation to the watch command. Instead of watching a microprocessor register or memory location, watch for a specific target program opcode value. This would be beneficial when trying to observe various system events. The command could be used similar to a general break point.

REFERENCES

1. Pierce, William H., unpublished communication, 1987.

2. Zumchak, E.M., Microcomputer Design and Troubleshooting, 1st. Ed., Howard W. Sams and Company, Indianapolis IN., 1982, page 192.

3. Commodore Business Machines Inc., Commodore 64 Programmer's Reference Guide, 1st. Ed., Howard W. Sams and Company, Indianapolis, IN. , 1983, page 402.

4. Osborne, A., Kane,G., Osborne's 4 & 8 bit Microprocessor Handbook, 1st. Ed., Osborne/McGraw-Hill, Berkley CA., 1981, page 10-24.

BIBLIOGRAPHY

Angerhausen, M., Becker, A., English, L., and Gerits, K., The Anatomy of the Commdore 64, 2nd. Ed., Abacus Software, Grand Rapids, MI., 1984.

Bywater. R.E., Hardware/Software Design of Digital Systems, 1st. Ed., Prentice Hall, NY., 1981.

Carson, J.H., Tutorial: Design of Microprocessor Systems, 1st. Ed., The Institute of Electrical and Electronic Engineers Inc, 1982.

Commodore Business Machines Inc., Commodore 64 Programmer's Reference Guide, 1st. Ed., Howard W. Sams and Company, Indianapolis, IN. , 1983.

Khosravi-Kamrani K., Toward a Low Cost MIcroprocessor Development System, Master of Engineering Thesis, Dept. of Electrical Engineering, University of Louisville, May, 1986.

Leventhal, L.A., 6502 Assembly Language Programming, 1st. Ed., Osborne/McGraw-Hill, Berkley CA., 1979.

Osborne, A., Kane,G., Osborne's 4 & 8 bit Microprocessor Handbook, 1st. Ed., Osborne/McGraw-Hill, Berkley CA., 181.

Osborne, A., Kane,G., Osborne's 16 bit Microprocessor Handbook, 1st. Ed., Osborne/McGraw-Hill, Berkley CA., 1981.

Osborne, A., Kane, G., An Introduction to Microcomputers, Vol. II, 1st. Ed., Osborne/McGraw-Hill, Berkley CA., 1979.

Pierce, William H., QK-02 Addressing Nomenclature, handwritten notes, 1984.

Texas Instruments Engineering Staff, The TTL Data Book, 1st. Ed., Texas Instruments Inc. Dallas, TX. 1976.

Zumchak, E.M., Microcomputer Design and Troubleshooting, 1st. Ed., Howard W. Sams and Company, Indianapolis IN., 1982.

APPENDIX A

PROTOTYPE HARDWARE DESIGN DATA

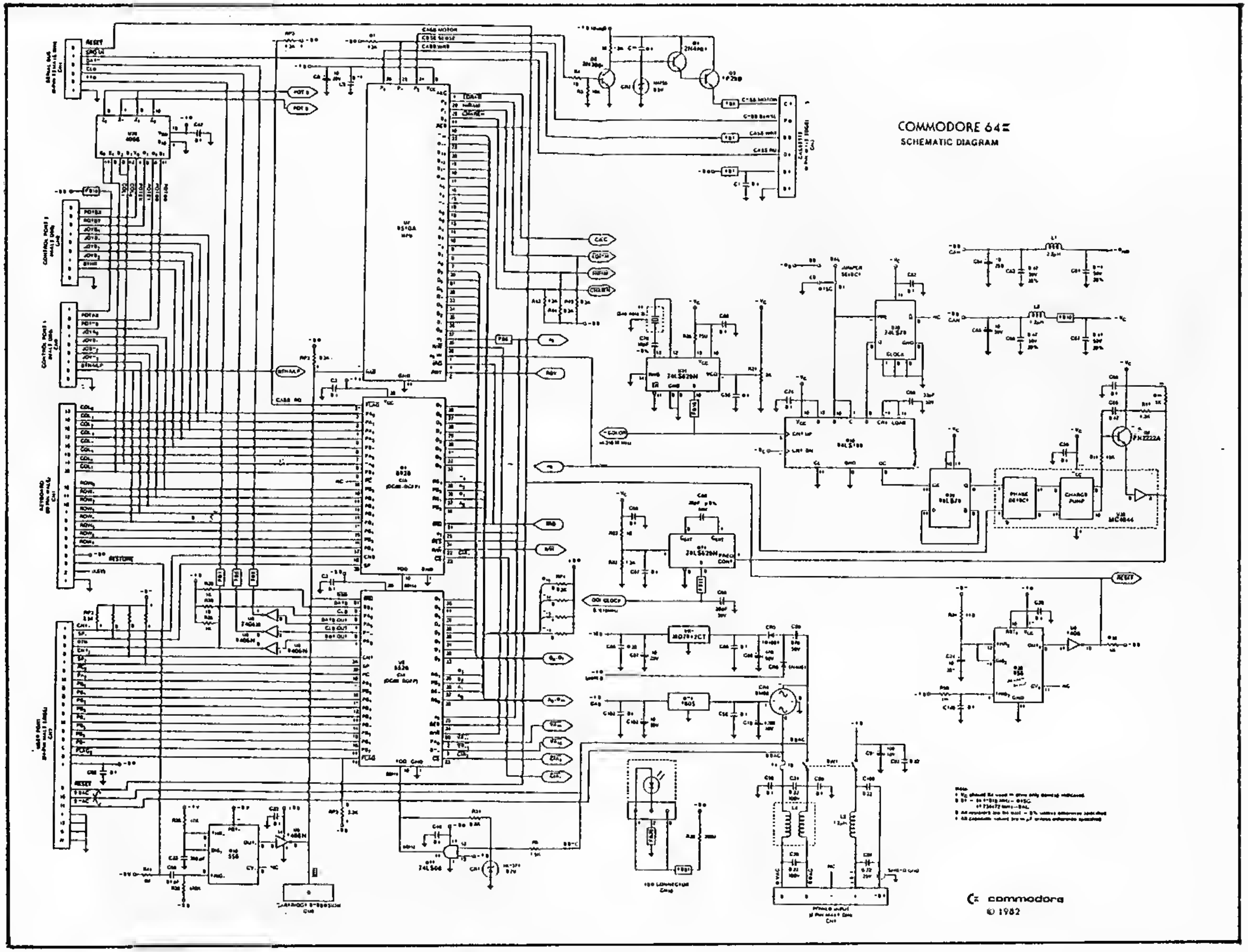


Figure 13 - Commodore 64 Schematic Diagram (part 1)

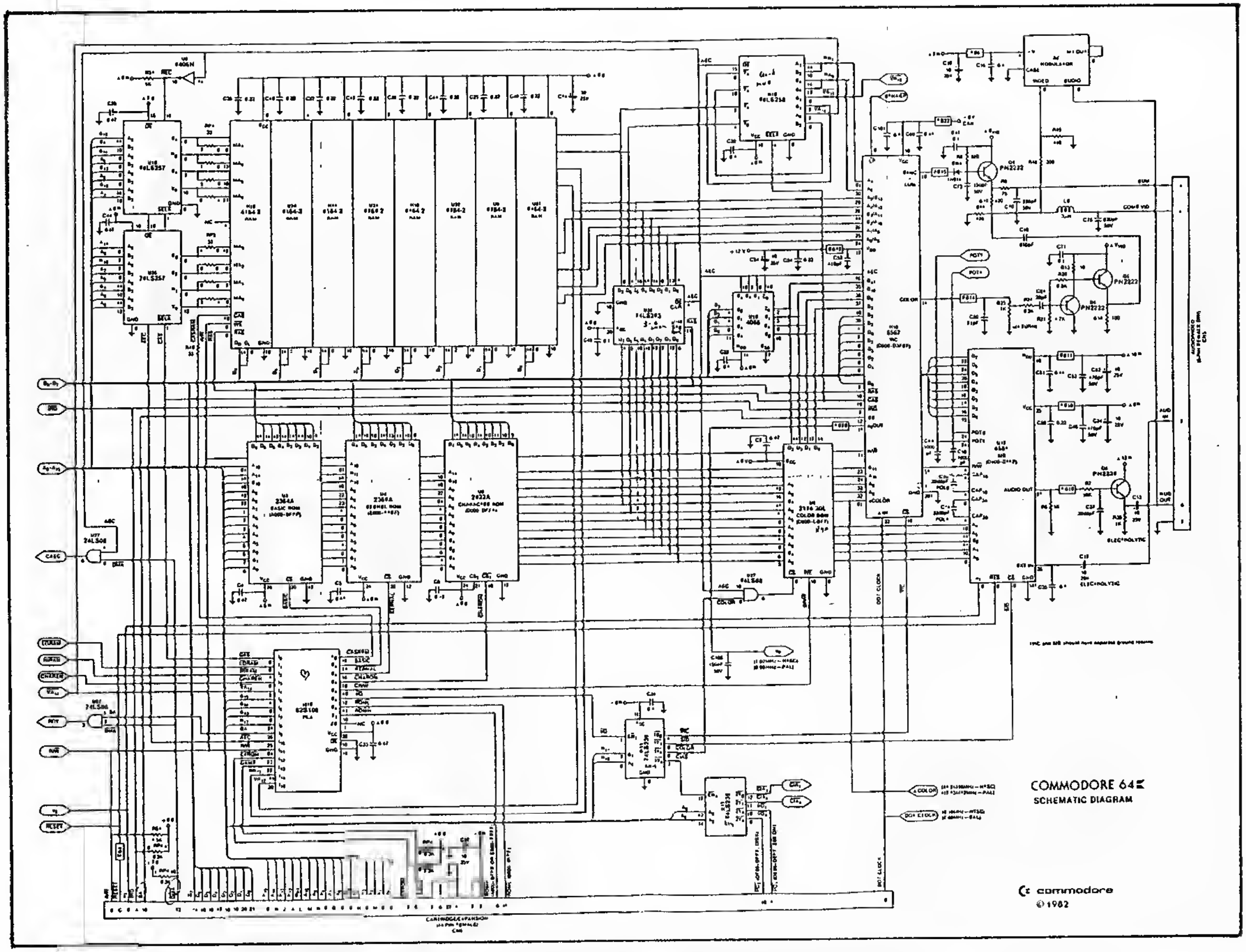


Figure 14 - Commodore 64 Schematic Diagram (part 2)

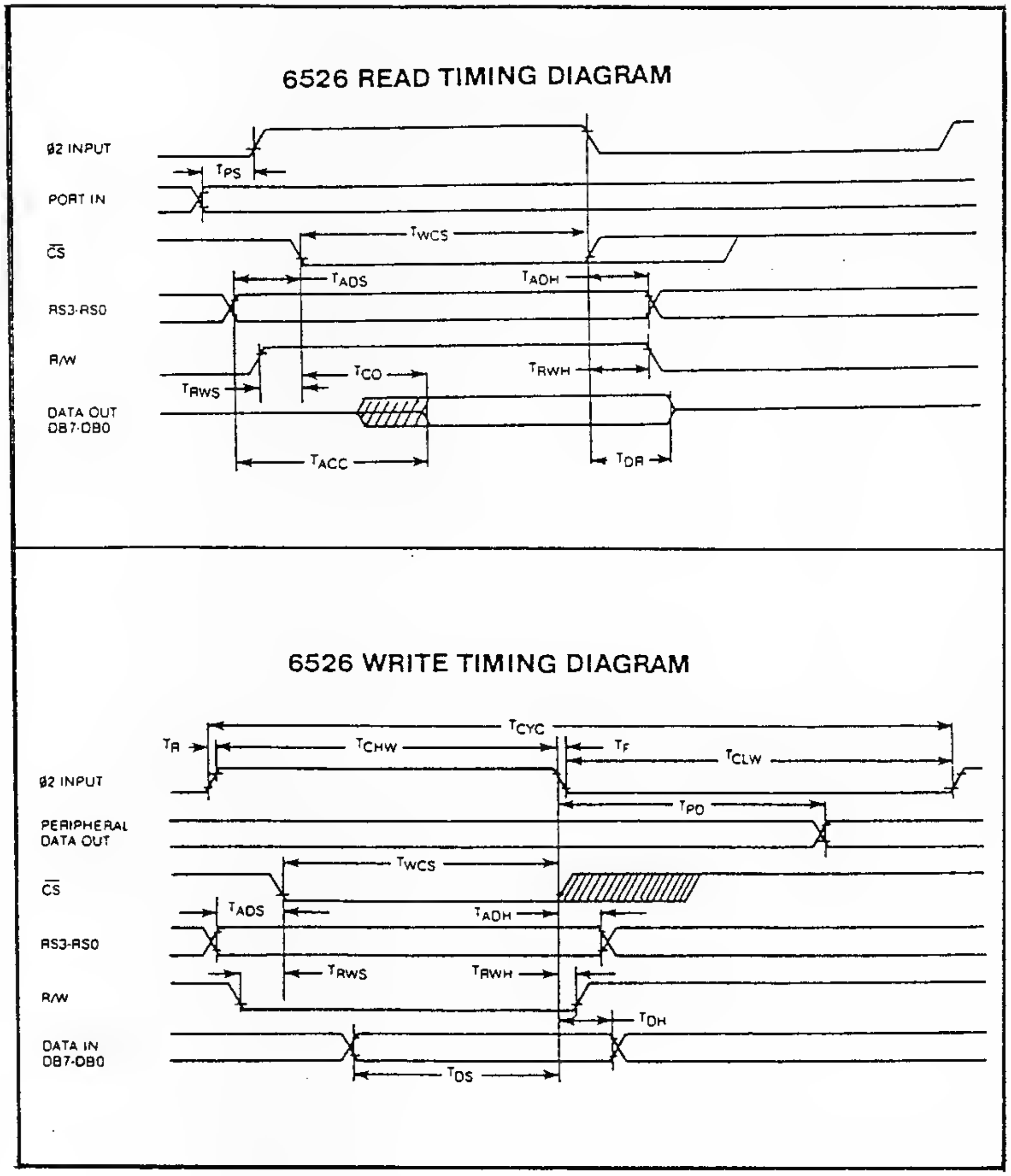


Figure 15 - 6526 CIA Read and Write Timing Diagram
(Reprinted from the C-64 Programmers Reference Guide)

Symbol	Characteristic	1MHz		2MHz		Unit
		MIN	MAX	MIN	MAX	
	$\phi 2$ Clock					
T_{CYC}	Cycle Time	1000	20,000	500	20,000	ns
T_R, T_F	Rise and Fall Time	—	25	—	25	ns
T_{CHW}	Clock Pulse Width (High)	420	10,000	200	10,000	ns
T_{CLW}	Clock Pulse Width (Low)	420	10,000	200	10,000	ns
	Write Cycle					
T_{PD}	Output Delay From $\phi 2$	—	1000	—	500	ns
T_{WCS}	$\overline{CS}$ low while $\phi 2$ high	420	—	200	—	ns
T_{ADS}	Address Setup Time	0	—	0	—	ns
T_{ADH}	Address Hold Time	10	—	5	—	ns
T_{RWS}	R/W Setup Time	0	—	0	—	ns
T_{RWH}	R/W Hold Time	0	—	0	—	ns
T_{DS}	Data Bus Setup Time	150	—	75	—	ns
T_{DH}	Data Bus Hold Time	0	—	0	—	ns
	Read Cycle					
T_{PS}	Port Setup Time	300	—	150	—	ns
T_{WCS}(2)	$\overline{CS}$ low while $\phi 2$ high	420	—	20	—	ns
T_{ADS}	Address Setup Time	0	—	0	—	ns
T_{ADH}	Address Hold Time	10	—	5	—	ns
T_{RWS}	R/W Setup Time	0	—	0	—	ns
T_{RWH}	R/W Hold Time	0	—	0	—	ns
T_{ACC}	Data Access from RS3-RS0	—	550	—	275	ns
T_{CO}(3)	Data Access from $\overline{CS}$	—	320	—	150	ns
T_{DR}	Data Release Time	50	—	25	—	ns

Figure 16 - 6526 Timing Characteristics
(Reprinted from the C-64 Programmers Reference Guide)

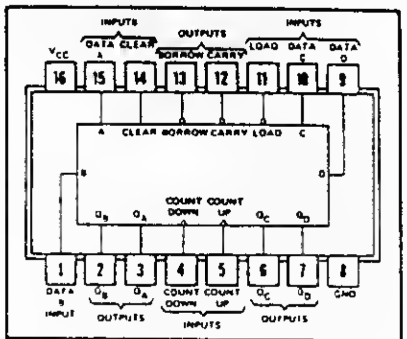
TYPES SN54192, SN54193, SN54L192, SN54L193, SN54LS192, SN54LS193 SN74192, SN74193, SN74L192, SN74L193, SN74LS192, SN74LS193 SYNCHRONOUS 4-BIT UP/DOWN COUNTERS (DUAL CLOCK WITH CLEAR)

BULLETIN NO. DL-S 7711828, DECEMBER 1972—REVISED AUGUST 1977

- Cascading Circuitry Provided Internally
- Synchronous Operation
- Individual Preset to Each Flip-Flop
- Fully Independent Clear Input

TYPES	TYPICAL MAXIMUM COUNT FREQUENCY	TYPICAL POWER DISSIPATION
'192, '193	32 MHz	325 mW
'L192, 'L193	7 MHz	43 mW
'LS192, 'LS193	32 MHz	95 mW

SN54', SN54LS' . . . J OR W PACKAGE
SN54L' . . . J PACKAGE
SN74', SN74L', SN74LS' . . . J OR N PACKAGE
(TOP VIEW)



logic: Low input to load sets $Q_A = A$, $Q_B = B$, $Q_C = C$, and $Q_D = D$

description

These monolithic circuits are synchronous reversible (up/down) counters having a complexity of 55 equivalent gates. The '192, 'L192, and 'LS192 circuits are BCD counters and the '193, 'L193 and 'LS193 are 4-bit binary counters. Synchronous operation is provided by having all flip-flops clocked simultaneously so that the outputs change coincidently with each other when so instructed by the steering logic. This mode of operation eliminates the output counting spikes which are normally associated with asynchronous (ripple-clock) counters.

The outputs of the four master-slave flip-flops are triggered by a low-to-high-level transition of either count (clock) input. The direction of counting is determined by which count input is pulsed while the other count input is high.

All four counters are fully programmable; that is, each output may be preset to either level by entering the desired data at the data inputs while the load input is low. The output will change to agree with the data inputs independently of the count pulses. This feature allows the counters to be used as modulo-N dividers by simply modifying the count length with the preset inputs.

A clear input has been provided which forces all outputs to the low level when a high level is applied. The clear function is independent of the count and load inputs. The clear, count, and load inputs are buffered to lower the drive requirements. This reduces the number of clock drivers, etc., required for long words.

These counters were designed to be cascaded without the need for external circuitry. Both borrow and carry outputs are available to cascade both the up- and down-counting functions. The borrow output produces a pulse equal in width to the count-down input when the counter underflows. Similarly, the carry output produces a pulse equal in width to the count-up input when an overflow condition exists. The counters can then be easily cascaded by feeding the borrow and carry outputs to the count-down and count-up inputs respectively of the succeeding counter.

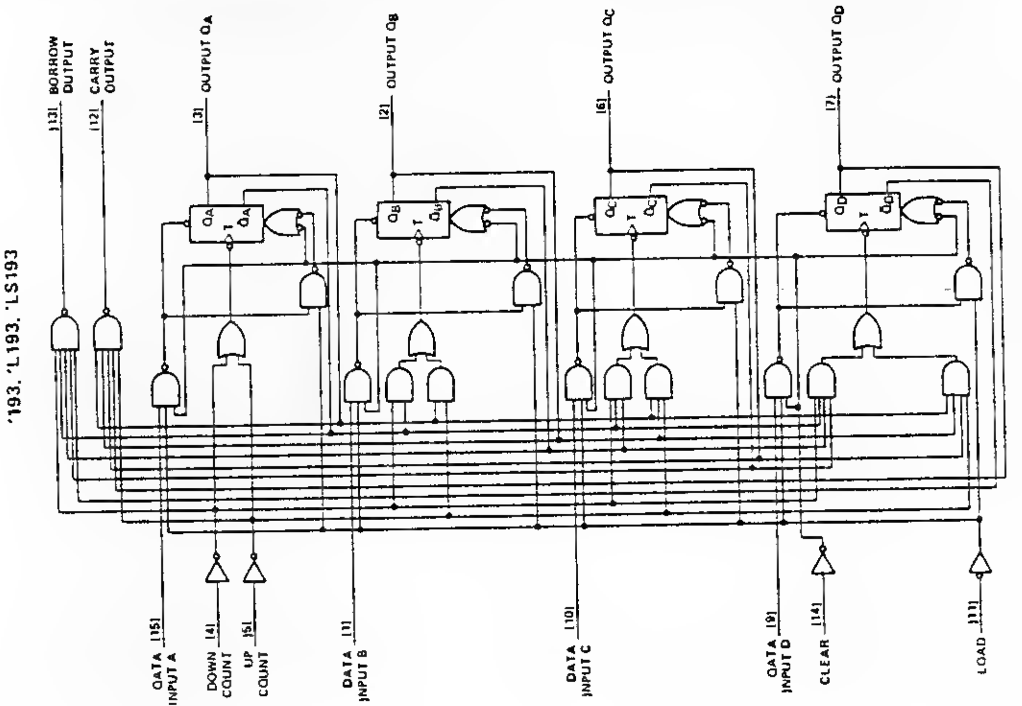
absolute maximum ratings over operating free-air temperature range (unless otherwise noted)

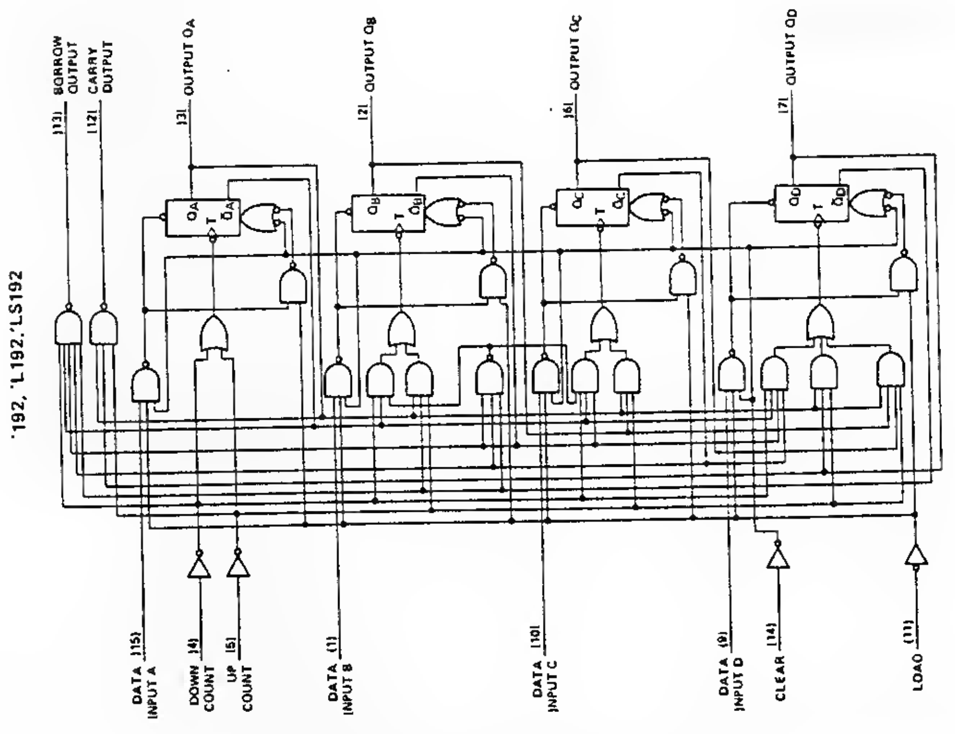
	SN54'	SN54L'	SN54LS'	SN74'	SN74L'	SN74LS'	UNIT
Supply voltage, V_{CC} (see Note 1)	7	8	7	7	8	7	V
Input voltage	5.5	5.5	7	5.5	5.5	7	V
Operating free-air temperature range		−55 to 125			0 to 70		°C
Storage temperature range		−65 to 150			−65 to 150		°C

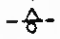
NOTE 1: Voltage values are with respect to network ground terminal.

TYPES SN54192, SN54193, SN54L192, SN54L193, SN54LS192, SN54LS193, SN74192, SN74193, SN74L192, SN74L193, SN74LS192, SN74LS193 SYNCHRONOUS 4-BIT UP/DOWN COUNTERS (DUAL CLOCK WITH CLEAR)

functional block diagrams





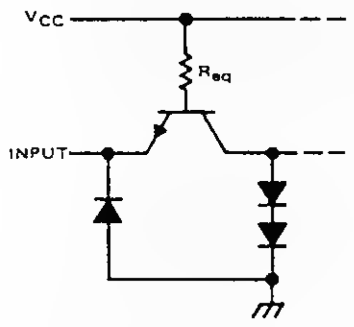
 . . . Dynamic input activated by a transition from a high level to a low level.

TYPES SN54192, SN54193, SN54L192, SN54L193, SN54LS192, SN54LS193, SN74192, SN74193, SN74L192, SN74L193, SN74LS192, SN74LS193 SYNCHRONOUS 4-BIT UP/DOWN COUNTERS (DUAL CLOCK WITH CLEAR)

REVISED OCTOBER 1976

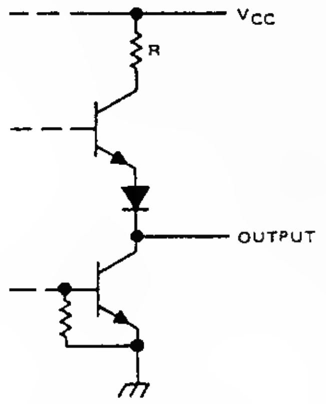
schematics of inputs and outputs

EQUIVALENT OF INPUTS
OF '192, '193, 'L192, 'L193



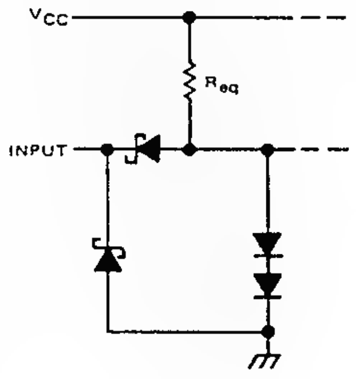
'192, '193: R_{eq} = 4 kΩ NOM
'L192, 'L193: R_{eq} = 40 kΩ NOM

TYPICAL OF OUTPUTS
OF '192, '193, 'L192, 'L193



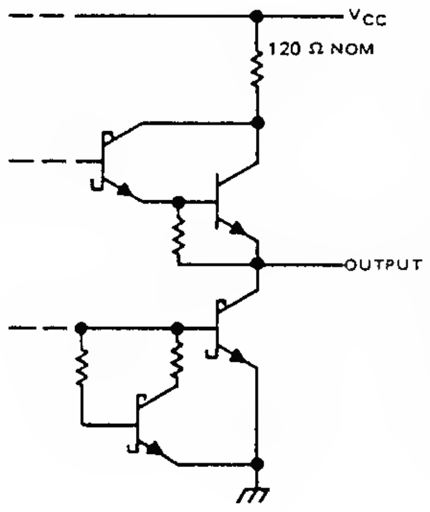
'192, '193: R = 130 Ω NOM
'L192, 'L193: R = 500 Ω NOM

EQUIVALENT OF INPUTS
OF 'LS192, 'LS193



Load Input: Req = 25 kΩ NOM
All other inputs: R_{eq} = 17 kΩ NOM

TYPICAL OF OUTPUTS
OF 'LS192, 'LS193



TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

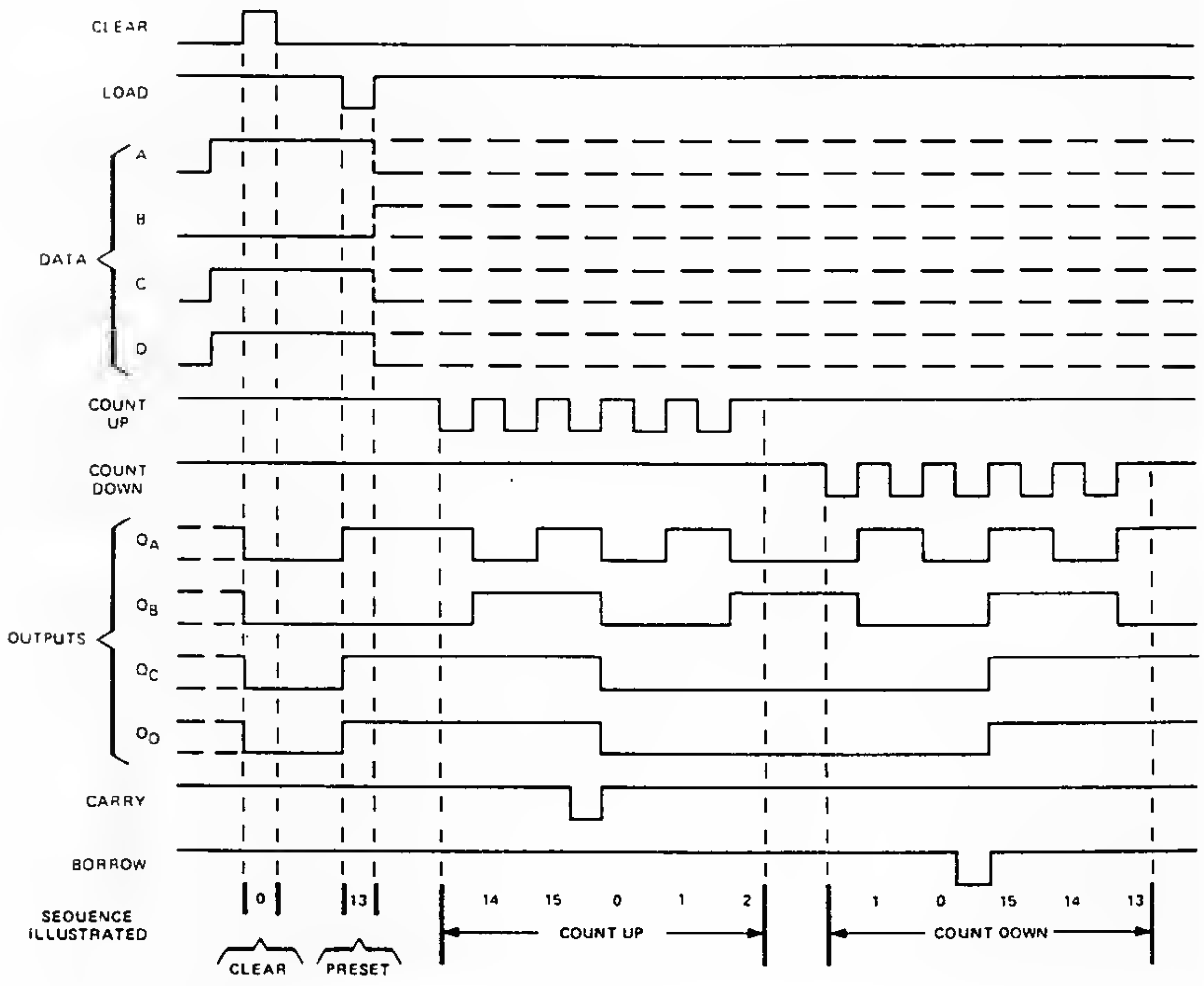
TYPES SN54193, SN54L193, SN54LS193, SN74193, SN74L193, SN74LS193 SYNCHRONOUS 4-BIT UP/DOWN COUNTERS (DUAL CLOCK WITH CLEAR)

'193, 'L193, 'LS193 BINARY COUNTERS

typical clear, load, and count sequences

Illustrated below is the following sequence:

1. Clear outputs to zero.
2. Load (preset) to binary thirteen.
3. Count up to fourteen, fifteen, carry, zero, one, and two.
4. Count down to one, zero, borrow, fifteen, fourteen, and thirteen.



NOTES: A. Clear overrides load, data, and count inputs.

B. When counting up, count-down input must be high; when counting down, count-up input must be high.

TYPES SN54192, SN54193, SN74192, SN74193 SYNCHRONOUS 4-BIT UP/DOWN COUNTERS (DUAL CLOCK WITH CLEAR)

recommended operating conditions

	SN54192 SN54193			SN74192 SN74193			UNIT
	MIN	NOM	MAX	MIN	NOM	MAX	
Supply voltage, V_{CC}	4.5	5	5.5	4.75	5	5.25	V
High-level output current, I_{OH}			−400			−400	μA
Low-level output current, I_{OL}			16			16	mA
Clock frequency, f_{clock}	0		25	0		25	MHz
Width of any input pulse, t_w	20			20			ns
Data setup time, t_{su} (see Figure 1)	20			20			ns
Data hold time, t_h	0			0			ns
Operating free-air temperature, T_A	−55		125	0		70	°C

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER		TEST CONDITIONS†	SN54192 SN54193			SN74192 SN74193			UNIT
			MIN	TYP‡	MAX	MIN	TYP‡	MAX	
V_{IH}	High-level input voltage		2			2			V
V_{IL}	Low-level input voltage				0.8			0.8	V
V_{IK}	Input clamp voltage	V_{CC} = MIN, I_I = −12 mA			−1.5			−1.5	V
V_{OH}	High-level output voltage	V_{CC} = MIN, V_{IH} = 2 V, V_{IL} = 0.8 V, I_{OH} = −400 μA	2.4	3.4		2.4	3.4		V
V_{OL}	Low-level output voltage	V_{CC} = MIN, V_{IH} = 2 V V_{IL} = 0.8 V, I_{OL} = 16 mA		0.2	0.4		0.2	0.4	V
I_I	Input current at maximum input voltage	V_{CC} = MAX, V_I = 5.5 V			1			1	mA
I_{IH}	High-level input current	V_{CC} = MAX, V_I = 2.4 V			40			40	μA
I_{IL}	Low-level input current	V_{CC} = MAX, V_I = 0.4 V			−1.6			−1.6	mA
I_{OS}	Short-circuit output current§	V_{CC} = MAX	−20		−65	−18		−65	mA
I_{CC}	Supply current	V_{CC} = MAX, See Note 2		65	89		65	102	mA

†For conditions shown as MIN or MAX, use the appropriate value specified under recommended operating conditions for the applicable type.
‡All typical values are at V_{CC} = 5 V, T_A = 25°C.
§Not more than one output should be shorted at a time.
NOTE 2: I_{CC} is measured with all outputs open, clear and load inputs grounded, and all other inputs at 4.5 V.

switching characteristics, V_{CC} = 5 V, T_A = 25°C

PARAMETER¶	FROM INPUT	TO OUTPUT	TEST CONDITIONS	MIN	TYP	MAX	UNIT
f_{max}			C_L = 15 pF, R_L = 400 Ω, See Figures 1 and 2	25	32		MHz
t_{PLH}	Count-up	Carry			17	26	ns
t_{PHL}					16	24	
t_{PLH}	Count-down	Borrow			16	24	ns
t_{PHL}					16	24	
t_{PLH}	Either Count	Q			25	38	ns
t_{PHL}					31	47	
t_{PLH}	Load	Q			27	40	ns
t_{PHL}					29	40	
t_{PHL}	Clear	Q			22	35	ns

¶f_{max} ≡ maximum clock frequency
t_{PLH} ≡ propagation delay time, low-to-high-level output
t_{PHL} ≡ propagation delay time, high-to-low-level output

TYPES SN54L192, SN54L193, SN74L192, SN74L193 SYNCHRONOUS 4-BIT UP/DOWN COUNTERS (DUAL CLOCK WITH CLEAR)

recommended operating conditions

	SN54L192 SN54L193			SN74L192 SN74L193			UNIT
	MIN	NOM	MAX	MIN	NOM	MAX	
Supply voltage, V_{CC}	4.5	5	5.5	4.75	5	5.25	V
High-level output current, I_{OH}			−100			−200	µA
Low-level output current, I_{OL}			2			3.6	mA
Clock frequency, f_{clock}	0		3	0		3	MHz
Width of any input pulse, t_w	200			200			ns
Data setup time, t_{su} (see Figure 1)	100			100			ns
Data hold time, t_h	0			0			ns
Operating free-air temperature range, T_A	−55		125	0		70	°C

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER		TEST CONDITIONS†	SN54L192 SN54L193			SN74L192 SN74L193			UNIT
			MIN	TYP‡	MAX	MIN	TYP‡	MAX	
V_{IH}	High-level input voltage		2			2			V
V_{IL}	Low-level input voltage				0.7			0.7	V
V_{IK}	Input clamp voltage	V_{CC} = MIN, I_I = −12 mA			−1.5			−1.5	V
V_{OH}	High-level output voltage	V_{CC} = MIN, V_{IH} = 2 V, V_{IL} = 0.7 V, I_{OH} = MAX	2.4	3.3		2.4	3.2		V
V_{OL}	Low-level output voltage	V_{CC} = MIN, V_{IH} = 2 V, V_{IL} = 0.7 V, I_{OL} = MAX		0.15	0.3		0.2	0.4	V
I_I	Input current at maximum input voltage	V_{CC} = MAX, V_I = 5.5 V			100			100	µA
I_{IH}	High-level input current	V_{CC} = MAX, V_I = 2.4 V			10			10	µA
I_{IL}	Low-level input current	V_{CC} = MAX, V_I = 0.3 V			−0.18			−0.18	mA
I_{OS}	Short-circuit output current§	V_{CC} = MAX	−3		−15	−3		−15	mA
I_{CC}	Supply current	V_{CC} = MAX, See Note 2		8.5	15		8.5	15	mA

†For conditions shown as MIN or MAX, use the appropriate value specified under recommended operating conditions for the applicable type.
‡All typical values are at V_{CC} = 5 V, T_A = 25°C.
§Not more than one output should be shorted at a time.
NOTE 2: I_{CC} is measured with all outputs open, clear and load inputs grounded, and all other inputs at 4.5 V.

switching characteristics, V_{CC} = 5 V, T_A = 25°C

PARAMETER¶	FROM INPUT	TO OUTPUT	TEST CONDITIONS	MIN	TYP	MAX	UNIT
f_{max}			C_L = 50 pF, R_L = 4 kΩ, See Figures 1 and 2	3	7		MHz
t_{PLH}	Count-up	Carry			65	130	ns
t_{PHL}					65	130	
t_{PLH}	Count-down	Borrow			65	130	ns
t_{PHL}					65	130	
t_{PLH}	Either Count	Q			104	200	ns
t_{PHL}					135	240	
t_{PLH}	Load	Q			130	240	ns
t_{PHL}					105	200	
t_{PHL}	Clear	Q			110	200	ns

¶f_{max} ≡ maximum clock frequency
t_{PLH} ≡ propagation delay time, low-to-high-level output
t_{PHL} ≡ propagation delay time, high-to-low-level output

APPENDIX B

PROGRAM LISTINGS

>>> FILE: DRIV.SO <<<

```
; ***************
; *             *
; * "DRIV.SO"   *
; *             *
; ***************
;
;
@$3000 ********** PROGRAM START **********
.     CLD
.     JSR-- $E544    ;CLEAR SCREEN
.     LDY#  $09
.     LDX#  $02
.     CLC
.     JSR-- $FFF0    ;SET CURSER
.     LDA#  $05
.     JSR-- $CF87    ;OUTPUT TITLE 1
.     LDY#  $0E
.     LDX#  $03
.     CLC
.     JSR-- $FFF0    ;SET CURSER
.     LDA#  $06
.     JSR-- $CF87    ;OUTPUT TITLE 2
.     JSR-- $CF82    ;SET FLAGS/REGISTERS
;
@$3021 ********** MAIN DEBUG LOOP **********
;
;       FROM THIS LOOP ALL COMMANDS ROUTINES ARE EXECUTED
;       SOME ROUTINES REFER TO THIS ROUTINE AS
;       THE MAIN DRIVER LOOP.
;
.     CLC
.     LDA#  $00
.     STA-- $033C    ;CLEAR ERROR FLAG
.     LDA#  $03      ;OUTPUT PROMPT
.     JSR-- $CF87    ;MESSAGE OUTPUT
.     JSR-- $CF7C    ;CLEAR INPUT BUFFER
.     JSR-- $CF7F    ;INPUT FROM KEYS
.     LDA#  $00
.     JSR-- $FFD2    ;CARRIAGE RETURN
.     LDA-- $0200
.     BEQ   *1       ;CHECK IF NO CMD
.     CMP#  $D1
.     BEQ   *2       ;GOTO 1 CHAR CMD
.     CMP#  $D2
.     BEQ   *3       ;GOTD 2 CHAR CMD
.     JMP-- *9       ;UNRECG CMD MESSAGE
*2    LDA#  $7E      ;LOW ADD OF 1CH TAB.
.     STA-  $19
.     LDA#  $C9      ;HIGH ADD OF 1CH TAB
.     STA-  $1A
```

>>> FILE: DRIV-SO <<<
(CONTINUED)

```
.     JMP--  *4
*3    LDA#   $AE      ;LOW ADD OF 2CH TAB.
.     STA-   $19
.     LDA#   $C9      ;HIGH ADD OF 2CH TAB
.     STA-   $1A
*4    LDY#   $00
*6    LDA)Y  $19      ;1/2 CH TABLE VECTOR
.     BNE    *5       ;END OF TABLE ?
*9    LDA#   $03      ;'UNRECOG CMD' MESSAGE
.     STA--  $033C    ;SET ERROR CODE
.     JSR--  $CF94    ;ERROR CHECK NORM RTS
.     JMP--  $CF73
*5    CMP--  $0201    ;ORIGINAL CMD VALUE
.     BEQ    *7       ;CMD FOUND !
.     INY
.     INY
.     INY
.     JMP--  *6
*7    INY             ;COPY COMMAND VECTOR
.     LDA)Y  $19      ; TO ZERO PAGE
.     STA-   $1B
.     INY
.     LDA)Y  $19
.     STA-   $1C
.     JSR--  *8       ;USE JSR TO ENTER
.     JMP--  $CF73    ;RETURN TO DRIVER
*8    JMP()  $001B    ;CMD VECTOR IS NOW
;                     ;IN 001B/001C
.     BRK
```

```
> > >   F I L E :   S U B 1 - S O   < < <

; ***********
; *         *
; *"SUB1.SO"*
; *         *
; ***********
;
;
@$8400 ********** MACHINE TO HEX OUTPUT **********
;
.     PHA
.     ROR
.     ROR
.     ROR
.     ROR
.     AND#  $0F
.     TAX
.     LDAXX $CF60
.     JSR-- $FFD2
.     PLA
.     AND#  $0F
.     TAX
.     LDAXX $CF60
.     JSR-- $FFD2
.     RTS
;
@$8419 ********** MACHINE TO BINARY OUTPUT **********
;
.     LDX#  $08
*1    PHA
.     AND#  $80
.     BEQ   *2
.     LDA#  $31      ; ASCII '1'
.     JMP-- *3
*2    LDA#  $30      ; ASCII '0'
*3    JSR-- $FFD2
.     PLA
.     ROL
.     DEX
.     BNE   *1
.     RTS
;
@$8430 ********** CLEAR INPUT BUFFER **********
;
.     LDA#  $00
.     LDX#  $00
*4    STAXX $0200
.     INX
.     CPX#  $5A
.     BNE   *4
.     RTS
```

>>> FILE: SUB1.SO <<<
(CONTINUED)

```
;
;********** ACCEPT INPUT FROM KEYBOARD ROUTINE  *********
;
@$843D
.     LDA#  $02
.     STA-  $1A
.     LDA#  $00
.     STA-  $19
*9    LDY#  $00
*10   JSR-- $E112
.     CMP#  $20     ;LEADING SPACES ?
.     BEQ   *18
.     CMP#  $2C     ;LEADING COMMAS ?
.     BEQ   *16
.     JMP-- *17
*5    JSR-- $E112
*17   CMP#  $22     ;QUOTED STRING ?
.     BEQ   *14
.     CMP#  $0D
.     BEQ   *6
.     CMP#  $20     ;SPACE DELIMITER ?
.     BEQ   *10
.     CMP#  $2C     ;COMMA DELIMITER ?
.     BEQ   *10
.     INY
.     STA)Y $19
.     JMP-- *5
*14   LDX#  $00
*15   JSR-- $E112
.     CMP#  $22
.     BEQ   *12
.     INX
.     STAXX $0230
.     JMP-- *15
*12   STX-- $0230
.     JMP-- *18
*10   TYA
.     LDY#  $00
.     STA)Y $19
.     LDA-  $19
.     CLC
.     ADC#  $05
.     STA-  $19
.     JMP-- *18
*6    TYA
.     LDY#  $00
.     STA)Y $19
.     RTS
;
;********** CK ERROR FLAG/OUTPUT MESSAGE **********
```

>>> FILE: SUB1.SO <<<
(CONTINUED)

```
;
@$B4B0
.     LDA--  $033C      ;GET ERROR FLAG
.     BEQ    *B         ;NO ERROR
.     ASL
.     TAX
.     PLA
.     PLA               ;WASTE FIRST RTS
.     JMP--  *13
.     LDA--  $033C
.     BEQ    *8
.     ASL
.     TAX
*13   LDAXX  $CC00
.     STA-   $20
.     INX
.     LDAXX  $CC00
.     STA-   $21
.     LDY#   $00
*7    LDA)Y  $20
.     BEQ    *8
.     JSR--  $FFD2
.     INY
.     JMP--  *7
*8    RTS
;
;********** SET SYSTEM FLAGS/REGISTERS/VECTORS **********
;
@$84F0
.     LDA#   $00
.     LDX#   $35     ;CLEAR WORKSPACE
*19   STAXX  $0338
.     DEX
.     BNE    *19
.     LDX#   $04
*11   LDAXX  $0315   ;SAVE BRK AND NMI VECTORS
.     STAXX  $035C
.     DEX
.     BNE    *11
.     LDX#   $FF
.     STX--  $DD03   ;SET DDR OF CIA 2
.     STX--  $DD01
;                     TO THE AUX CIRCUIT.
.     LDA#   $A0
.     STA--  $0316   ;BREAK LOW BYTE
.     LDA#   $B6
.     STA--  $0317   ;BREAK HIGH BYTE
.     LDA#   $B0
.     STA--  $0318   ;NMI LOW BYTE
.     LDA#   $97
```

```
>>> FILE: SUB1.SO <<<
              (CONTINUED)

.     STA-- $0319   ;NMI HIGH BYTE
.     LDA#  $7F
.     STA-  $38     ;PROTECT MEMORY-T
.     LDA#  $FF
.     STA-  $37     ;PROTECT MEMORY-T
.     LDA#  $02
.     STA-- $0350   ;CLEAR BREAK PT CODE
.     STA-- $0351
.     LDA#  $80
.     STA-- $0354   ;SET DELAY COUNTER
.     RTS
```

>>> FILE: SUB2.SO <<<

```
; ***********
; *         *
; *"SUB2.SO"*
; *         *
; ***********
;
;
;********** ASCII TO HEX CONVERSION  **********
;
@$8590
.     INY
.     STY-  $14      ;FINAL LOCATION
.     LDAXX $0200    ;GET #OF CHARS
.     CMP#  $02      ; 2 CHARS
.     BEQ   *9
.     CMP#  $04      ; 4 CHARS
.     BEQ   *8
      LDA#  $02      ; SYNTAX ERROR
.     STA-- $033C
.     RTS
*8    INX
.     JSR-- *7       ;CONVERT HIGH BYTE
*10   LDY-  $14      ;GET HIGH LOCATION
.     STAYY $0000    ;STORE HIGH BYTE
.     DEC-  $14      ;GET LOW LOCATION
.     INX            ;PT TO LOW CHARS
.     JSR-- *7       ;CONVERT LOW BYTE
.     LDY-  $14      ;GET LOW LOCATION
.     STAYY $0000    ;STORE LOW BYTE
.     RTS
*9    LDA#  $00
.     JMP-- *10
;
; 2 CHAR ASCII TO HEX INTERNAL VALUE
;
*7    LDAXX $0200       ;FIRST CHARACTER
.     JSR-- *2          ;CONVERT
.     LDA-- $033C       ;CHECK FOR ERROR
.     BNE   *1
.     TYA               ;MOVE INTO ACC.
.     ROL
.     ROL
.     ROL
.     ROL
.     AND#  $F0
.     STA-- $0352       ;TEMP STORAGE
.     INX
.     LDAXX $0200       ;SECOND CHAR
.     JSR-- *2          ;CONVERT
.     LDA-- $033C       ;CHECK FOR ERROR
```

>>> FILE: SUB2.SO <<<
(CONTINUED)

```
.     BNE    *1
.     TYA                   ;MOVE INTO ACC.
.     ORA-- $0352           ;COMBINE VALUES
*1    RTS
;
; ASCII CHARACTER SEARCH ROUTINE
;
*2    LDY#   $00
*3    CMP-Y $CF60           ;CMP WITH HEX TAB
.     BEQ    *4
.     INY
.     CPY#   $10            ;TRY ALL VALUES?
.     BEQ    *5
.     JMP-- *3
*4    RTS
*5    LDA#   $01            ;SET ERROR FLAG
.     STA-- $033C
.     RTS
;
;********** ADD TO ZP INDEX 'A' **********
; INDEX 'A' IS LOCATED AT $19/$1A
;
@$2500
.     STA-   $26
.     CLC
.     LDA-   $19
.     ADC-   $26
.     STA-   $19
.     LDA-   $1A
.     ADC#   $00
.     STA-   $1A
.     RTS
;
;********** ADD TO ZP INDEX 'B' **********
; INDEX 'B' IS LOCATED AT $1B/$1C
;
@$2510
.     STA-   $26
.     CLC
.     LDA-   $1B
.     ADC-   $26
.     STA-   $1B
.     LDA-   $1C
.     ADC#   $00
.     STA-   $1C
.     RTS
;
;********** ADD TO ZP INDEX 'C' **********
; INDEX 'C' IS LOCATED AT $1D/$1E
;
```

> > > FILE: SUB2.SO < < <

(CONTINUED)

```
@$8620
.   . STA-   $26
.     CLC
.     LDA-   $10
.     ADC-   $26
.     STA-   $1D
.     LDA-   $1E
.     ADC#   $00
.     STA-   $1E
.     RTS
;
;*********** COMPARE ZP INDEXES 'A' TO 'B' **********
;  NOTE...
;          IF A<B   THEN CARRY IS CLEARCC
;          IF A>=B THEN CARRY IS SET
@$8630
.     SEC
.     LDA-   $19
.     SBC-   $1B
.     STA-   $25
.     LDA-   $1A
.     SBC-   $1C
.     ORA-   $25
.     RTS
;
;*********** MISC SCREEN FORMATTING ROUTINES *********
;
@$8640
.     LDA#   $20    ;OUTPUT A SPACE
.     JSR-- $FFD2
.     RTS
;
@$8646
.     LDA#   $0D    ;OUTPUT A CARRIAGE RETUURN
.     JSR-- $FFD2
.     RTS
```

```
>>> FILE: SUB3.SO <<<


;***********
;*         *
;*"SUB3.SO"*
;*         *
;***********
;
;
@$8S50 ********** OUTPUT REGISTERS **********
;
.     LDA#  $13
.     STA-  $D3
.     LDA-- $0342  ;ACC VALUE
.     JSR-- $CF76  ;ASCII HEX OUTPUT
.     JSR-- $CFB5  ;<SP>
.     LDA-- $0341  ;X REGISTER
.     JSR-- $CF76
.     JSR-- $CF23
.     LDA-- $0340  ;Y REGISTER
.     JSR-- $CF76
.     JSR-- $CFB5
.     LDA-- $0346  ;STACK POINTER
.     JSR-- $CF76
.     JSR-- $CFB5
.     LDA-- $0343  ;PROCESSOR STAT WORD
.     JSR-- $CF79  ;ASCII BINARY OUTPUT
.     JSR-- $CFB8  ;<CR>
.     RTS
;
```

>>> FILE: CMD1-SO <<<

```
; ***********
; *         *
; *"CMD1.SO"*
; *         *
; ***********
;
;
@$9120 ********** DISPLAY MEMORY COMMAND ROUTINE **********
;
 .     LDY#   $19        ;PNTR TO FINAL LOC
 .     LDX#   $05        ;PNTR WITHIN INBUFF
 .     JSR--  $CF65      ;CONVERT
 .     JSR--  $CF31      ;ERROR CHECK-NORTS
 .     LDA--  $020F
 .     BNE    *2
 .     LDA#   $00    ;SETS UP FOR 1 LINE
 .     STA-   $1B    ; - OF OUTPUT IF THE
 .     STA-   $1C    ; - SECOND AD IS NUL
 .     JMP--  *4
*2     LDY#   $1B        ;PNTR TO FINAL LOC
 .     LDX#   $0A        ;PNTR WITHIN INBUFF
 .     JSR--  $CF65      ;CONVERT
 .     JSR--  $CF31      ;ERROR CHECK-NORTS
*4     LDA#   $00
 .     JSR--  $FFD2
 .     LDA#   $01
 .     JSR--  $CFA0
*1     JSR--  $CFB5
 .     LDY#   $00
 .     LDA-   $1A
 .     JSR--  $CF76
 .     LDA-   $19
 .     JSR--  $CF76
 .     LDA#   $3A
 .     JSR--  $FFD2
*3     JSR--  $CFB5
 .     LDA)Y  $19
 .     JSR--  $CF76
 .     INY
 .     CPY#   $08
 .     BNE    *3
 .     JSR--  $CFB5
 .     LDY#   $00
*6     LDA)Y  $19
 .     CMP#   $60
 .     BCS    *7
 .     CMP#   $20
 .     BCS    *5
*7     LDA#   $2D
*5     JSR--  $FFD2
```

>>> FILE: CMD1.SO <<<
(CONTINUED)

```
.     INY
.     CPY#  $08
.     BNE   *6
.     JSR-- $CFB8
.     LDA#  $08
.     JSR-- $CF9D   ;ADD 8 TO INDEX 'A'
.     JSR-- $FFE4   ;CHECK IF KEY PRESSED
.     CMP#  $20     ;SPACE ?
.     BNE   *13
*8    JSR-- $FFE4   ;START INTO HOLD LOOP
.     BEQ   *8
*13   CMP#  $51     ;'Q'-QUIT
.     BEQ   *12
*9    JSR-- $CFAE   ;COMPARE 'A' TO 'B'
.     BCC   *1
*12   JSR-- $CFB8
.     RTS
;
@#91AA ********** HIDE SUBROUTINE COMMAND ROUTINE *********
;
.     LDA-  $25     ;GET LAST OPCODE
.     CMP#  $20     ;WAS IT 'JSR'
.     BNE   *14     ;ERROR MESSAGE
.     LDA#  $01
.     STA-- $033D   ;SET SUB SUFFRESS FLAG
.     STA-- $0357   ;SET SUB LEVEL COUNTER
.     STA-- $034B   ;SET TRACE MODE
.     STA-- $0353   ;SET INSTRUCTION COUNTER
.     STA-- $0359   ;SET HIDE SUB FLAG
.     PLA           ;ADJUST STACK
.     PLA
.     JMP-- $CF88   ;GOTO TRACE DRIVER
*14   LDA#  $04     ;ERROR MESSAGE
.     STA-- $033C   ;SET ERROR FLAG
.     JSR-- $CF91   ;NO RTS FROM THIS ROUTINE.
;
@#92EE ********** FILL MEMORY COMMAND ROUTINE **********
;
.     LDY#  $19     ;START LOCATION
.     LDX#  $05     ;BUFFER POINTER
.     JSR-- $CF85   ;CONV ASCII TO HEX
.     LDY#  $1B     ;END LOCAT/OPERAND
.     LDX#  $0A     ;BUFFER POINTER
.     JSR-- $CF85   ;CONVERT
.     JSR-- $CF91
.     LDY#  $00
.     LDA-  $1C     ;CHECK IF 1 FILL
.     BNE   *10
.     LDA-  $1B     ;SINGLE LOC FILL
.     STA)Y $19
```

```
>>> FILE: CMD1-SO <<<
          (CONTINUED)

.     RTS
;MULTI FILL SECTION
*10   LDY#  $10
.     LDX#  $0F
.     JSR-- $CF85
.     JSR-- $CF91    ;CHECK FOR ERROR
.     LDA#  $01
.     JSR-- $CFA0
.     LDY#  $00
*11   LDA-  $10
.     STA)Y $19
.     LDA#  $01
.     JSR-- $CF9D    ;ADD 1 TO INDEX 'A'
.     JSR-- $CFA6    ;COMPARE 'A' TO 'B'
.     BCC   *11
.     RTS
```

>>> FILE: CMD2.SO <<<

```
;***********
;*         *
;*"CMD2.SO"*
;*         *
;***********
;
;
;**** GK-02 DISASSEMBLY COMMAND ROUTINE *********
;
@$3220
.     LDA-- $0205  ;CHECK IF NO PARAMS
.     BEQ   *12
.     LDY#  $10    ;PNTR LOCATION
.     LDX#  $05    ;BUFFER LOCATION
.     JSR-- $CF85  ;CONVERT
.     JSR-- $CF31  ;ERROR CHECK NO-RTS
*12   JSR-- $E544  ;CLEAR SCREEN
.     JSR-- $CFB8
.     LDX#  $15    ;LINE COUNTER (20)
.     STX-  $1F    ;HOLDS # OF LINES
*1    LDA#  $2E    ; "."
.     JSR-- $FFD2
.     LDA-  $1E
.     JSR-- $CF76
.     LDA-  $1D
.     JSR-- $CF76
.     LDX#  $03
*14   JSR-- $CFB5
.     DEX
.     BNE   *14
.     JSR-- $CFAC  ;ACTUAL DISASMBL ROUT
.     JSR-- $CFB8
.     DEC-  $1F
.     BNE   *1
.     JSR-- $CF28
.     RTS
;
;*** DISASSEMBLY SUBROUTINE ***
;
@$3330
.     LDA#  $09    ;LO ADD OF OCT TABLE
.     STA-  $1B
.     LDA#  $C3    ;HI ADD OF OCT TABLE
.     STA-  $1C
.     LDY#  $00
.     STY-  $20    ;ML MEMONIC POINTER
*2    LDA)Y $1B
.     CMP#  $02    ;END OF OPCODE TABLE ?
.     BNE   *8
.     LDX#  $03    ;SEARCH FOR OPCODE FAILED
```

```
>>>  FILE:  CMD2.SO  <<<
                (CONTINUEO)

.    LDA#  $2A    ; "*"
*9   JSR-- $FF02
.    DEX
.    BNE   *9
.    LDA#  $01    ;INCREMENT INOEX 'C'
.    JSR-- $CFA3
.    RTS
*8   CMP#  $FF    ;CHECK IF NEW MEMONIC
.    BNE   *11    ;NO - NOT A NEW MEMON
.    INC-  $20    ;SET FOR NEXT MEMONIC
.    LDA#  $01
.    JMP-- *4
*11  CMP)Y $1C    ;COMP WITH CURRENT
.    BEQ   *3     ;OPCOOE MATCH
.    LDA#  $02    ;STEP THROUGH OP TABLE
*4   JSR-- $CFA0  ;ADD ACC TO INDEX 'B'
.    JMP-- *2
*3   CLC
.    LDA-  $20    ;TRIPLE MEMONIC VALUE
.    ADC-  $20    ;   "
.    ADC-  $20    ;   "
.    TAX
.    LDAXX $CB41  ;START OF MEMON TABLE-1
.    JSR-- $FF02  ;OUTPUT 1ST CHAR OF MEMONIC
.    INX
.    LOAXX $CB41
.    JSR-- $FF02  ;OUTPUT 2NO CHAR OF MEMONIC
.    INX
.    LOAXX $CB41
.    JSR-- $FFD2  ;OUTPUT 3RD CHAR OF MEMONIC
.    INY
.    LDA)Y $1B    ;GET ADO/MDD VALUE
.    STA-  $1A    ;TEMP STORE IT
.    ANO#  $0F    ;MASK BYTE = 00001111
.    STA-  $19    ;TEMP STORE IT
.    CLC
.    ADC-  $19    ;1/2 OF OFST-AOO TABL
.    TAX
.    LDAXX $CBE9  ;START DF AOD TABLE
.    JSR-- $FFD2  ;DUTPUT 1ST CHAR AODR
.    INX
.    LDAXX $CBE9
.    JSR-- $FF02  ;DUTFUT 2NO CHAR ADDR
.    JSR-- $CFB5
.    LDA-  $1A
.    AND#  $40    ;MASK BYTE = 01000000
.    BEQ   *5     ;NOT RELATIVE BRANCH
.    LDA#  $24
.    JSR-- $FF02
.    LDA)Y $1D
```

>>> FILE: CMD2.SO <<<
(CONTINUED)

```
.     STA-  $19
.     ROL
.     BCC   *16
.     LDA#  $FE     ;BACKWARD BRANCH
.     SBC-  $19
.     STA-  $19
.     SEC
.     LDA-  $1D
.     SBC-  $19
.     STA-  $19
.     LDA-  $1E
.     SBC#  $00
.     JMP-- *15
*16   INC-  $19     ;FORWARD BRAHCH
.     INC-  $19
.     LDA-  $19
.     AND#  $7F
.     ADC-  $1D
.     STA-  $19
.     LDA-  $1E
.     ACC#  $00
*15   JSR-- $CF76
.     LDA-  $19
.     JSR-- $CF76
.     LDA#  $02
.     JMP-- *13
*5    LDA-  $1A     ;*** NORMAL EXTRA BYTE ROUTE ***
.     ROR
.     ROR
.     ROR
.     ROR
.     AND#  $03     ;MASK BYTE = 00000011
.     TAY
.     STY-  $1A
.     BEQ   *7
.     LDA#  $24     ; "$"
.     JSR-- $FFD2
*6    LDA)Y $1D     ;OUTPUT EXTRA BYTES
.     JSR-- $CF76
.     DEY
.     BNE   *6
*7    INC-  $1A
.     LDA-  $1A
*13   JSR-- $CFA3
.     RTS
```

>>> FILE: CMD3.SO <<<

```
;***********
;*         *
;*"CMD3.SO"*
;*         *
;***********
;
;
@$9090 ********** LOAD MEMORY COMMAND ROUTINE **********
;
.     LDA-- $0205  ;RELOCATED LOAD ?
.     BEQ   *2
.     LDY#  $19    ;RELO LOAD ROUTINE
.     LDX#  $05
.     JSR-- $CF65  ;CONVERT ASCII-HEX
.     JSR-- $CF91  ;ERROR CHECK ND RTS
.     LDA#  $00
.     STA-  $26
.     JMP-- *3
*2    LDA#  $01    ;STANDARD LOAD ROUTINE
.     STA-  $26
*3    LDA#  $06    ;LOGICAL FILE #
.     LDX#  $08    ;DEVICE-DISK DRIVE
.     LDY-  $26    ;SECONDARY ADDRESS
.     JSR-- $FFBA  ;SET FILE SPEC'S
.     LDA-- $0230  ;LENGTH OF FILENAME
.     LDX#  $31    ;LOW ADDR OF FILENAME
.     LDY#  $02    ;HIGH ADDR OF FILENAME
.     JSR-- $FFBD  ;SET FILENAME SPEC'S
.     LDX-  $19    ;LOW ADD PROG START
.     LDY-  $1A    ;HIGH ADDR PROG START
.     LDA#  $00    ;LOAD FLAG
.     JSR-- $FFD5  ;LOAD ROUTINE
.     JMP-- *1
;
@$90D0 ********** SAVE MEMORY COMMAND ROUTINE **********
;
.     LDY#  $13    ;PNTR TO START ADDR
.     LDX#  $05    ;PNTR IN INPUT BUFFER
.     JSR-- $CF85  ;CONVERT
.     JSR-- $CF91  ;ERROR CHECK-ND RTS
.     LDY#  $1B    ;PNTR TO END ADDR
.     LDX#  $0A    ;PNTR IN INPUT BUFFER
.     JSR-- $CF85  ;CONVERT
.     JSR-- $CF91  ;ERROR CHECK-ND RTS
.     LDA#  $01
.     JSR-- $CFA0  ;INCREMENT INDEX 'B'
.     LDA#  $01    ;LOGICAL FILE #
.     LDX#  $08    ;DEVICE-DISK DRIVE
.     LDY#  $00    ;SECONDARY ADDRESS
.     JSR-- $FFBA  ;SET FILE SPEC'S
```

>>> FILE: CMD3.SO <<<
(CONTINUED)

```
.     LDA-- $0230  ;LENGTH OF FILE NAME
.     LDX#  $31    ;LOW ADDR OF FILENAME
.     LDY#  $02    ;HIGH ADDR OF FILENAME
.     JSR-- $FFBD  ;SET FILE NAME
.     LDA#  $1B    ;ADDR OF START
.     LDX-  $1B    ;LOW END ADDR+1
.     LDY-  $1C    ;HIGH END ADDR+1
;                   $1B/$1C HAS START ADD
.     JSR-- $FFD8  ;SAVE ROUTINE
*1    JSR-- $CF88  ;OUTPUT <CR>
.     LDA#  $03
.     JSR-- $CF97  ;OUTPUT 'OK'
.     RTS
;
@$9410 ********* DISPLAY REGISTER COMMAND ROUTINE  *********
;
.     LDA-- $0202  ;GET 2ND CMD LETTER
.     CMP#  $52    ;'R'
.     BEQ   *4
.     LDA#  $03
.     STA-- $033C  ;SET ERROR #
.     JSR-- $CF91  ;ERROR CHECK NO-RTS
*4    LDA#  $1E    ;SPACE OVER
.     STA-  $03    ;COLUMN STORAGE
.     LDA#  $08    ;MESS #
.     JSR-- $CF97  ;MESSAGE OUTPUT
.     LDA#  $13    ;MESS #
.     JSR-- $CF97  ;OUTPUT PC VALUE
.     LDA-- $0345
.     JSR-- $CF76
.     LDA-- $0344
.     JSR-- $CF76
.     JSR-- $CF85  ;DISPLAY REG ROUTINE
.     RTS
;
@$9443 ********** EXIT COMMAND ROUTINE *********
;
.     LDX#  $04
*5    LDAXX $035C  ;GET BRK & NMI VECTOR
.     STAXX $0315
.     DEX
.     BNE   *5
.     JSR-- $CF9A  ;CLEAR ANY BREAKPOINTS
.     PLA
.     PLA
;     RTS          ;RETURN TO SYSTEM
.     JMP-- $FE43  ;BASIC WARM START VECTOR
;
@$9460 ********** SET BREAK POINT COMMAND **********
;
```

>>> FILE: CMD3.SO <<<
(CONTINUED)

```
.     LDA--  $0202  ;GET 2ND CHARACTER
.     SEC
.     SBC#   $31    ;ADJUST VALUE
.     STA-   $26    ;TEMP STORAGE
.     BEQ    *6
.     CMP#   $01
.     BEQ    *6
.     LDA#   $03    ;'UNRECOG CMD' MESSAGE
*8    STA--  $033C  ;SET ERROR FLAG
.     JSR--  $CF91  ;ERROR CHECK NO RTS
*6    LDY-   $26
.     LDAYY  $0350  ;CHECK IF ALREADY SET
.     CMP#   $02    ;CODE FOR NOT SET
.     BEQ    *7
.     LDA#   $04    ;'CANNOT EXECUTE' MESSAGE
.     JMP--  *8
*7    LDY#   $19
.     LDX#   $05
.     JSR--  $CF85  ;CONVERT
.     JSR--  $CF91  ;ERROR CHECK NO RTS
.     LDY-   $26
.     LDX#   $00
.     LDAX)  $19
.     STAYY  $0350  ;STORE VALUE UNDER BREAK
.     LDA#   $00
.     STAX)  $19    ;INSERT ACTUAL '00'
.     LDA-   $19
.     STAYY  $034C  ;STORE LOW BYTE
.     LDA-   $1A
.     STAYY  $034E  ;STORE HIGH BYTE
.     LDA#   $09    ;'OK' MESSAGE
.     JSR--  $CF87
.     RTS
```

> > >   F I L E :   C M D 4 . S O   < < <

```
;***********
;*         *
;*"CMD4.SO"*
;*         *
;***********
;
;
2$3480 ********** CLEAR ... COMMAND ROUTINES **********
;         - CLEAR SUBROUTINE SUPPRESSION
;         - CLEAR WATCH PARAMETER
;         - CLEAR BREAK POINTS
;
.     LDA-- $0202  ;GET 2ND CHARACTER
.     CMP#  $42    ;'B' FOR CLEAR BREAK
.     BNE   *4
@$3487 *** SUBR ENTRY POINT
.     LDX#  $00
*2    LDAXX $0350  ;GET STORED VALUE
.     BEQ   *19    ;PASS IF BRK EXISTS
.     CMP#  $02    ;CHECK IF NOT SET
.     BEQ   *3
.     LDAXX $034C  ;GET LOW BYTE
.     STA-  $19
.     LDAXX $034E  ;GET HIGH BYTE
.     STA-  $1A
.     LDAXX $0350  ;GET  OPCODE
.     LDY#  $00
.     STA)Y $19    ;REPLACE OPCODE
*19   LDA#  $02
.     STAXX $0350  ;CLEAR CODE
*3    INX
.     CPX#  $02    ;CHECK IF DONE
.     BNE   *2
.     JMP-- *6
;
*4    CMP#  $53    ;'S' FOR CLEAR SUBROUTINE SUPPRESSION
.     BNE   *5
.     LDA#  $00
.     STA-- $0330  ;CLEAR FLAG
.     JMP-- *6
;
*5    CMP#  $57    ;'W' FOR CLEAR WATCH
.     BNE   *20
.     LDA#  $00
.     STA-- $033E  ;WATCH FLAG
*6    LDA#  $09    ;'OK' MESSAGE
.     JSR-- $CF97
.     RTS
*20   CMP#  $52    ;'R' - CLEAR REGISTERS
.     BNE   *7
```

>>> FILE: CMD4.SO <<<
(CONTINUED)

```
.     LDA#  $00
.     STA-- $0340  ;CLEAR Y REGISTER
.     STA-- $0341  ;CLEAR X REGISTER
.     STA-- $0342  ;CLEAR ACCUMULATOR
.     BEQ   *6
;
*7    LDA#  $03    ;'UNRECOG CMD' MESSAGE
.     JSR-- $CF97
.     RTS
;
@$3515 ********** PARAMETER COMMAND ROUTINE *********
;
.     LDX#  $00
.     STX-  $26
*8    LDA#  $0C    ;MESSAGE #
.     JSR-- $CF97
.     LDA-  $26
.     CLC
.     ADC#  $31
.     JSR-- $FFD2
.     LDX-  $26
.     LDAXX $0350
.     CMP#  $02
.     BNE   *10
.     LDA#  $10
.     JSR-- $CF97
      JMP-- *9
*10   LDA#  $11
.     JSR-- $CF97
.     LDX-  $26
.     LDAXX $034E
.     JSR-- $CF76
.     LDX-  $26
.     LDAXX $034C
.     JSR-- $CF76
.     JSR-- $CF88
*9    INC-  $26
.     LDA-  $26
.     CMP#  $02
.     BNE   *8
.     LDA#  $0D
.     JSR-- $CF97
.     LDA-- $033D
.     BEQ   *11
.     LDA#  $0F
.     BNE   *12
*11   LDA#  $10
*12   JSR-- $CF97  ;DISPLAY WATCH VALUE
.     LDA#  $0E
.     JSR-- $CF97
```

>>> FILE: CMD4.SO <<<
(CONTINUED)

```
.     LDA-- $033E
.     BNE   *13
.     LDA#  $10
.     JSR-- $CF97
.     RTS
*13   CMP#  $01
.     BNE   *14
.     LDA#  $53
.     BNE   *17
*14   CMP#  $02
.     BNE   *15
.     LDA#  $58
.     BNE   *17
*15   CMP#  $03
.     BNE   *16
.     LDA#  $41
.     BNE   *17
*16   LDA#  $24
.     JSR-- $FFD2
.     LDA-- $034A
.     JSR-- $CF76
.     LDA-- $0349
.     JSR-- $CF76
.     JMP-- *16
*17   JSR-- $FFD2
*19   LDA#  $12
.     JSR-- $CF97
.     LDA-- $033E   ;GET WATCH FLAG
.     CMP#  $05     ;SEE IF WATCH WORD
.     BNE   *21
.     LDA#  $24
.     JSR-- $FFD2   ;OUTPUT '$'
.     LDA-- $035B
.     JSR-- $CF76
.     LDA-- $035A
.     JMP-- *22
*21   LDA-- $033F   ;GET WATCH VALUE
*22   JSR-- $CF76
.     JSR-- $CFB8
.     RTS
```

>>> FILE: CMD5.SO <<<

```
;***********
;*         *
;*"CMD5.SO"*
;*         *
;***********
;
;
@$95E3 ********** GO COMMAND ROUTINE **********
;
.     LDY#  $19     ;PNTR TO FINAL LOCATION
.     LDX#  $05     ;PNTR TO ORIGIN
.     JSR-- $CF35   ;CONVERT ASCII
.     JSR-- $CF91   ;ERROR CHECK - NO RTS
.     PLA
.     PLA
.     LDY-- $0340   ;LOAD Y REGISTER
.     LDX-- $0341   ;LOAD X REGISTER
.     LDA-- $0342   ;LOAD ACCUMULATOR
.     JMP() $0019   ;TARGET
;
@$9602 ********** LOAD/WATCH COMMAND ROUTINE **********
;
.     LDA#  $01     ;WATCH CMD ENTRY POINT
.     STA-  $24     ;TEMP FLAG
.     BNE   *10     ;ALWAYS BRANCH
@$9608
.     LDA#  $00
.     STA-  $24
*10   LDY#  $19
.     LDX#  $05
.     JSR-- $CF85   ;CONVERT ASCII
.     JSR-- $CF91   ;ERROR CHECK - NO RTS
.     LDA#  $00
.     STA-  $25     ;TEMP POINTER
.     LDA-- $0202   ;GET 2ND CHARACTER
.     CMP#  $59     ;Y-REG WATCH OR LOAD
.     BEQ   *3
.     CMP#  $58     ;X-REG WATCH OR LOAD
.     BEQ   *2
.     CMP#  $41     ;ACC WATCH OR LOAD
.     BEQ   *1
.     CMP#  $50     ;LOAD PROGRAM COUNTER
.     BEQ   *4
.     CMP#  $4D     ;WATCH MEMORY LOCATION
.     BEQ   *7
.     CMP#  $57     ;'W'
.     BEQ   *7
*6    LDA#  $03     ;UNRECOG CMD MESSAGE
.     STA-- $033C
.     JSR-- $CF91   ;ERROR CHECK - NO RTS
```

```
>>> FILE: CMD5.SO <<<
                (CONTINUED)

*1    INC-   $25
*2    INC-   $25
*3    INC-   $25
.     LDA-   $24     ;DETERMINE IF WATCH OR LOAD
.     BNE    *5      ;GOTO WATCH Y X ACC SECTION
.     LDA-   $19     ;LOAD Y X ACC SECTION
.     LDX-   $25
.     STAXX  $033F
*9    LDA#   $03     ;OUTPUT 'OK' MESSAGE
.     JSR--  $CF97
.     RTS
*4    LDA-   $24     ;CHECK IF LOAD ONLY
.     BNE    *6      ;GOTO UNRECOG CMD MESSAGE
.     LDA-   $19     ;LOAD PROGRAM COUNTER SECTION
.     STA--  $0344   ;LOW BYTE
.     LDA-   $1A
.     STA--  $0345   ;HIGH BYTE
.     JMP--  *9
*5    LDA-   $25     ;WATCH Y X ACC SECTION
*11   STA--  $033E   ;SET WATCH FLAG
*8    LDA-   $19
.     STA--  $033F   ;STORE VALUE TO WATCH FOR
.     JMP--  *9
*7    LDA-   $24     ;CHECK IF WATCH ONLY
.     BEQ    *6      ;GOTO UNRECOG CMD MESSAGE
.     LDA-   $19
.     STA--  $0349   ;LOW BYTE OF MEMORY LOCATION
.     LDA-   $1A
.     STA--  $034A   ;HIGH BYTE OF MEMORY LOCATION
.     LDY#   $19
.     LDX#   $0A
.     JSR--  $CFB5   ;CONVERT ASCII
.     JSR--  $CF91   ;ERROR CHECK - NO RTS
.     LDA--  $0202   ;GET 2ND CHAR AGAIN
.     CMP#   $57     ; W'
.     BEQ    *15
.     LDA#   $04     ;WATCH MEM FLAG
.     BNE    *11
*15   LDA-   $19
.     STA--  $035A   ;SET WORD WATCH LOW BYTE
.     LDA-   $1A
.     STA--  $035B   ;SET WORD WATCH HIGH BYTE
.     LDA#   $05
.     BNE    *11     ;GOTO SET WATCH FLAG
;
@$9CAA ******** SET SUBROUTINE SUPPRESSION COMMAND ********
;
.     LDA#   $01
.     STA--  $033D ;SET FLAG
.     BNE    *9
```

```
>>>  FILE:  CMD5.SO  <<<
                (CONTINUED)

;
@$96B2 ********* TRACE/EXECUTE SINGLE STEP ROUTINE *******
;
;       THIS ROUTINE SETS UP THE PARAMETERS FOR
;       THE SINGLE STEPPING ROUTINES.  SOME ROUTINES
;       REFER TO THIS ROUTINE AS THE TRACE ORIVER.
;
.     LDA#   $01     ;TRACE ENTRY POINT
.     JMP--  *12
;
@$96B8
.     LDA#   $00     ;NO TRACE ENTRY POINT
*12   STA--  $034E   ;SET/CLEAR OISPLAY FLAG
.     LDX--  $0202   ;GET 2ND CHARACTER - IF PRESENT
.     STX--  $0353   ;STORE IN TRACE CONTROL REGISTER
.     CPX#   $42     ;CHECK FOR "BREAK" VARIATION
.     BNE    *13
.     LDX#   $02
*14   LDA,X  $034F   ;GET BREAK FLAG
.     CMP#   $02     ;SEE IF SET
.     SNE    *13
.     DEX
.     BNE    *14     ;CHECK NEXT BREAKPOINT
.     LDA#   $0C     ;OUTPUT PART OF MSG
.     JSR--  $CF37
.     DEC-   $D3     ;BACKSPACE ON SCREEN
.     LDA#   $10     ;OUTPUT PART OF MSG
.     JSR--  $CF37
.     RTS
*13   LDY--  $0205   ;CHECK IF NO COUNT SPECIFIED
.     BNE    *18
.     INY
.     JMP--  *19
*18   LDY#   $19     ;GET VALUE FROM KEYS
.     LDX#   $05
.     JSR--  $CF25
.     JSR--  $CF91
.     LDY-   $19
*19   STY--  $0355   ;SET INSTRUCTION COUNTER
.     STY--  $0352   ;SET BREAK COUNTER
.     LDA#   $00
.     STA--  $0357   ;RESET SUBR LEVEL
.     STA--  $0353   ;RESET BRK PT OFFSET
.     PLA            ;ADJUST STACK DUE TO JSR
.     PLA
.     LOA--  $034B
.     BEQ    *20
.     LDA#   $0A
.     JSR--  $CF37
.     LDA#   $12
```

>>> FILE: CMD5.SO <<<

(CONTINUED)

```
.     STA-  $D3
.     LDA#  $0B
.     JSR-- $CF97
;
;    PREFAIR FOR NMI GENERATION ...
;
K20   SEI              ;BLOCK KEY SCANS
.     LDA-- $0345  ;GET HIGH BYTE OF INST
.     PHA
.     LDA-- $0344  ;GET LOW BYTE OF INST
.     PHA
.     LDA-- $0343  ;GET PSW
.     PHA
.     LDY-- $0340  ;GET Y REGISTER
.     LDA-- $0342  ;GET ACCUMULATOR
.     LDX#  $00
.     STX-- $DD01  ;START AUX COUNTER CIRCUIT
.     INX
.     STX-- $DD01  ;END OF PULSE TO AUX CIRCUIT
.     LDX-- $0341  ;GET X REGISTER
.     RTI              ;PICK UP AT PROPER POINT
;                       IN THE TARGET PROGRAM
```

```
>>>   FILE:  CMD6.SO   <<<


;************
;*          *
;*"CMD6.SO"*
;*          *
;************
;
;
:$0780 ********** NMI SINGLE STEPPING ROUTINE **********
;       THIS ROUTINE IS CALLED UPON VIA THE NMI
;       VECTOR.
;
.     STY-- $0340  ;STORE Y REGISTER
.     STX-- $0341  ;STORE X REGISTER
.     STA-- $0342  ;STORE ACC
.     PLA          ;GET PSW
.     STA-- $0343  ;STORE PSW
      PLA          ;GET LOW BYTE OF NEXT INST
.     STA-- $0347  ;STORE
.     PLA          ;GET HIGH BYTE OF NEXT INST
.     STA-- $0348  ;STORE
.     TSX          ;MOVE STACK POINTER
.     STX-- $0346  ;STORE STACK POINTER
.     CLI
.     LDA-- $0344  ;GET CURRENT OPCODE LOCATION
.     STA-  $1D
.     LDA-- $0345
.     STA-  $1E
.     LDY#  $00
.     LDA)Y $1D    ;GET CURRENT OPCODE
.     STA-  $25    ;TEMP STORE OPCODE
.     BNE   *31
.     JMP-- $CF8E  ;GO TO BRK ENTRY POINT IN BRK ROUTINE
*31   LDA-- $0357  ;CHECK SUBR LEVEL COUNTER
.     BNE   *33
.     DEC-- $0355  ;DECREMENT INSTRUCTION COUNTER
.     LDA-- $034B  ;CHECK DISPLAY/NODISPLAY FLAG
.     BEQ   *33
.     LDX-- $0354  ;START SCREEN DELAY TIMER
*13   LDY#  $FF    ;DELAY IS APPROX ($0354)*0.0013
*14   DEY
.     BNE   *14
.     DEX
.     BNE   *13
.     INC-  $D3    ;THIS SECTION OUTPUTS ONE
.     LDA-  $1E    ; LINE OF SCREEN DISPLAY.
.     JSR-- $CF78  ;OUTPUT HIGH MEMORY BYTE
.     LDA-  $1D
.     JSR-- $CF78  ;OUTPUT LOW MEMORY BYTE
.     INC-  $D3
.     JSR-- $CFAC  ;OUTPUT DISASSEMBLED MEMONIC
```

>>> FILE: CMD6.SO <<<
(CONTINUED)

```
.     JSR--  $CF2B   ;OUTPUT REGISTERS
*33   LDA--  $0358   ;GET BRK JUST HIT FLAG
.     BEQ    *9      ;NOT SET
.     LOY--  $0353   ;GET BRK POINT OFFSET
.     LDAYY  $034B   ;GET LOW ADD OF BRK PT
.     STA-   $19
.     LDAYY  $0340   ;GET HIGH ADD OF BRK PT
.     STA-   $1A
.     LDA#   $00
.     TAY
.     STA)Y  $19     ;RE-INSERT THE '00' BRK PT
.     STA--  $0358   ;CLEAR BRK JUST HIT FLAG
*9    LDX--  $033E   ;CHECK IF WATCH SET
.     BEQ    *1      ;$03 = NOT SET
.     CPX#   $04     ;CHECK IF WATCH MEMORY
.     BEQ    *2
.     CPX#   $05     ;CHECK IF WATCH WORD
.     BEQ    *2
.     LOAXX  $033F   ;GET X
*3    CMP--  $033F   ;COMPARE TO INPUT WATCH VALUE
.     BNE    *1
*37   LDA#   $0E     ;WATCH FULFILLED MSG #
.     JSR--  $CF97   ;OUTPUT MESSAGE
.     LDA#   $14
.     JSR--  $CF97
.     JMP--  *13     ;FINISH UP ROUTINE
*2    LOA--  $0348   ;GET MEMORY LOCATION
.     STA-   $19
.     LDA--  $034A
.     STA-   $1A
.     LDA--  $033E   ;GET WATCH FLAG
.     CMP#   $05     ;CHECK IF WATCH WORO
.     BEQ    *34
.     LDY#   $00
.     LDA)Y  $19     ;GET MEMORY VALUE
.     JMP--  *3      ;GO TO COMPARE
*34   LOY#   $00
*38   LDA)Y  $19
.     CMPYY  $035A
.     BNE    *1
.     INY
.     CPY#   $02
.     BNE    *38
.     BEQ    *37
*1    JSR--  $FFE4   ;GET CHARACTER FROM SYSTEM INPUT BUFFER
.     BEQ    *4
.     CMP#   $11     ;CSR ON - SLOWER
.     BNE    *21
.     LOA#   $08     ;INCREMENT VALUE
.     AOC--  $0354   ;ADD TO TIMER VALUE
```

>>> FILE: CMD6-SO <<<
(CONTINUED)

```
.     JMP--  *35
*21   CMP#   $91      ;CSR UP - FASTER
.     BNE    *22
.     LDA--  $0354    ;GET TIMER VALUE
.     SBC#   $08      ;DECREMENT VALUE
*35   STA--  $0354    ;STORE TIMER VALUE
.     JMP--  *4
*22   CMP#   $20      ;HOLD
.     BNE    *10
*5    JSR--  $FFE4    ;HOLD LOOP
.     BEQ    *5
*10   CMP#   $51      ;'Q' QUIT
.     BNE    *4
.     JMP--  *16      ;GO TO MAIN DRIVER LOOP
*4    LDA-   $25      ;GET CURRENT OPCODE VALUE
*27   LDX--  $034B    ;GET TRACE/EXECUTE FLAG
.     BEQ    *7
.     LDX--  $0330    ;GET SUB SUPRESS FLAG
.     BEQ    *7
.     CMP#   $20      ;TEST FOR 'JSR'
.     BNE    *6
.     INC--  $0357    ;INCREMENT SUBR LEVEL COUNTER
.     JMP--  *7
*6    CMP#   $60      ;TEST FOR 'RTS'
.     BNE    *7
.     DEC--  $0357    ;DECREMENT SUBR LEVEL
.     BNE    *7
.     JSR--  $CF6B    ;OUTPUT REGISTERS
*7    LDX--  $0356    ;GET TRACE CONTROL REGISTER
.     BEQ    *17
*15   CPX#   $43      ;'C' CONTINOUS TRACE
.     BEQ    *17      ;VALIDATE THE NEXT INSTRUCTION
.     CPX#   $42      ;'B' CONTROLLED BREAK
.     BEQ    *17
.     CPX#   $4A      ;'J' GO UNTIL JUMP
.     BEQ    *36
.     JMP--  *13      ;GOTO ERROR ROUTINE
*36   CMP#   $4C      ;'JMP--' OPCODE
.     BEQ    *23      ;STOP EXECUTION
.     CMP#   $6C      ;'JMP()' OPCODE
.     BEQ    *23      ;STOP EXECUTION
.     AND#   $1F      ;MASK (TEST FOR BRANCH)
.     CMP#   $10      ;TEST FOR BRANCH
.     BEQ    *23
*17   LDA--  $0357    ;GET SUB LEVEL
.     BNE    *25
.     LDA--  $034B    ;GET TRACE/EXECUTE FLAG
.     BEQ    *25
.     LDA--  $0347    ;GET NEXT OPCODE
.     STA-   $1B
```

>>> FILE: CMD6.SO <<<
(CONTINUED)

```
.     LDA--  $0348
.     STA-   $1C
.     LDX#   $D9     ;SET OPCODE TABLE VECTORS
.     STX-   $19
.     LDX#   $C9
.     STX-   $1A
.     LDY#   $00
*26   LDA)Y  $19     ;GET TEST OPCODE FROM TABLE
.     CMP#   $02     ;END OF TABLE ?
.     BEQ    *24     ;-THEN INVALID OPCODE
.     CMP#   $FF
.     BNE    *23
.     LDA#   $01
.     BNE    *30
*23   CMP)Y  $1E     ;COMPARE TO NEXT MEMONIC
.     BEQ    *25     ;VALID OPCODE!
.     LDA#   $02     ;STEP THROUGH TABLE
*30   JSR--  $CF3D   ;ADD ACC TO INDEX 'A'
.     JMP--  *26
*25   LDA--  $0347   ;GET LOW BYTE OF NEXT OPCODE
.     STA--  $0344   ;MOVE TO CURRENT HIGH OPCODE
.     LDA--  $0348   ;GET HIGH BYTE OF NEXT OPCODE
.     STA--  $0345   ;MOVE TO CURRENT HIGH OPCODE
.     LDA--  $0357   ;GET SUB LEVEL
.     BNE    *32
.     LDA--  $0356   ;GET TRACE CONTROL REGISTER
.     BNE    *32
.     LDA--  $0355   ;GET INSTRUCTION COUNTER
.     BNE    *32
.     JMP--  *16     ;GO TO MAIN DRIVER LOOP
*32   JMP--  $CF86   ;GO TO TRACE DRIVER
*24   LDA#   $15     ;"INVALID OPCODE" MSG #
.     JSR--  $CF97
.     LDA-   $1C
.     JSR--  $CF76
.     LDA-   $1B
.     JSR--  $CF76
.     JMP--  *28
*20   LDA#   $16     ;"JMP OR BRANCH DETECTED" MSG #
*11   JSR--  $CF97
.     LDA--  $0345
.     JSR--  $CF76
.     LDA--  $0344
.     JSR--  $CF76
*28   JSR--  $CFB8
*16   LDA--  $0359   ;HIDE SUB FLAG
.     BEQ    *39
.     LDA#   $00
.     STA--  $0359   ;CLEAR HIDE SUB FLAG
.     STA--  $0330   ;CLEAR SUB SUPPRESS FLAG
```

```
>>> FILE: CMD6.SO <<<
           (CONTINUED)

*39   JMP--  $CF73   ;MAIN DRIVER LOOP
*18   LDA#   $03
.     STA--  $033C
.     JSR--  $CF94   ;ERROR CHECK - NO RTS
.     JMP--  $CF73
```

```
> > >   F I L E :   B R K 1 . S O   < < <

; ***********
; *         *
; *"BRK1.SO"*
; *         *
; ***********
;
@$35E0 ********** BREAK POINT HANDLING ROUTINE **********
;
;  BRK (SOFTWARE INTERRUPT) ENTRY POINT
;
.     SEI
.     PLA              ;Y REG
.     STA-- $0340
.     PLA              ;X REG
.     STA-- $0341
.     PLA              ;ACCUMULATOR
.     STA-- $0342
.     PLA
.     STA-- $0343   ;PSW
.     PLA              ;SHOULD BE PC LOW
.     SEC
.     SBC#  $02      ;CORRECT
.     STA-- $0344   ;STORE PC LOW
.     PLA              ;SHOULD BE PC HIGH
.     SBC#  $00      ;CORRECT
.     STA-- $0345
.     TSX              ;TRANSFER STACK POINTER
.     STA-- $0346   ;STORE STACK POINTER
;
@$3602  ENTRY POINT WHEN '00' OPCODE DETECTED DURING NMI
;        SINGLE STEP ROUTINE.
;
.     LDA-- $0358   ;GET TRACE CONTROL FLAG
.     CMP#  $42      ;'B' N BREAKS
.     BEQ   *1
*4    LDA#  $12
.     STA-  $03      ;SET COLUMN
.     LDA#  $0B      ;MESSAGE #
.     JSR-- $CF87
.     LDA#  $07
.     JSR-- $CF87   ;MESSAGE
.     LDA-- $0345   ;GET HIGH BYTE OF ADD
.     JSR-- $CF76   ;INT TO HEX OUT
.     LDA-- $0344   ;GET LOW BYTE OF ADD
.     JSR-- $CF76
.     JSR-- $CF8B   ;DISPLAY REG ROUTINE
.     CLI
.     JMP-- $CF73   ;DRIVER LOOP
*1    DEC-- $0352   ;DECREMENT BRK COUNTER
.     BEQ   *4
```

>>> FILE: BRK1-SO <<<
(CONTINUED)

```
.     LDA#  $01
.     STA-- $0358   ;SET BRK JUST HIT FLAG
.     LDY#  $02
*2    LDA-- $0344   ;GET CURRENT LOW ADD
.     STA-  $19
.     CMPYY $034C   ;CHECK AGAINST BRK LOW
.     BNE   *3
.     LDA-- $0345   ;GET CURRENT HIGH ADD
.     STA-  $1A
.     CMPYY $034D   ;CHECK AGAINST BRK HIGH
.     BEQ   *5
*3    DEY
.     BNE   *2
.     BEQ   *4
*5    STY-- $0353   ;STORE OFFSET TO BRK PT
.     LDX#  $00
.     LDAYY $034F   ;GET OPCODE VALUE
.     STAX) $19     ;REPLACE '00' WITH OPCODE VALUE
.     LDA-- $0343   ;GET PSW
.     AND#  $EF     ;CLEAR BREAK FLAG
.     STA-- $0343
.     LDA-- $0348  ;SCREEN DISPLAY MODE
.     BEQ   *6
.     LDA-- $0357  ;GET SUB LEVEL COUNTER
.     BNE   *6
.     LDA#  $2B    ;'+'
.     JSR-- $FFD2  ;OUTPUT A CHARACTER
.     DEC-  $D3
*6    JMP-- $CF88   ;JUMP TO TRACE DRIVER
;                    INSTRUCTION EXECUTED WILL BE THE
;                    OPCODE UNDER THE BREAK POINT.
```

```
>>>  FILE:  MESS.SO  <<<

; ***********
; *         *
; *"MESS.SO"*
; *         *
; ***********
;
;
;       THIS FILE CONTAINS ALL THE ASCII MESSAGES
;       GENERATED BY THE VARIOUS ROUTINES.
;
@$CC50
A          HEX ERROR
T          $0D $0D $00
A          COMMAND SYNTAX ERROR
T          $0D $0D $00
A          UNRECOGNIZED COMMAND
T          $0D $0D $00
A          CANNOT EXECUTE COMMAND
T          $0D $0D $0D $00
A          GK-02 DEBUGGER/TRACER
T          $0D $00
A          VERSION 3.1
T          $0D $00
A          BREAK AT $
T          $00
A          >
T          $00
A          OK
T          $0D $00
A            PC   MEMONIC
T          $00
A          ACC XR YR SP NV*BDIZC
T          $0D $00
A          BREAKPOINT #
T          $00
A          SUBROUTINE SUPRESSION
T          $00 $00
A          WATCH
T          $00
A           SET
T          $0D $00
A           NOT SET
T          $0D $00
A           SET AT $
T          $00
A           FOR $
T          $00
A             PC=$
T          $00
A            COMPLETE
```

>>> FILE: MESS.SO <<<
(CONTINUED)

```
T            $0D $00
A            INVALID OPCODE AT $
T            $00
A            JMP OR BRANCH AT $
T            $00
;
;***** MESSAGE LINK TABLE *****
;
T$CC02       $80 $CC $8C $CC $A3 $CC
T            $BA $CC $D4 $CC $EB $CC
T            $F8 $CC $03 $CD $05 $CD
T            $08 $CD $13 $CD $2F $CD
T            $3C $CD $53 $CD $5A $CD
T            $68 $CD $6A $CD $74 $CD
T            $72 $CD $82 $CD $8D $CD
T            $A1 $CD
```

>>> FILE: JUMP.TBL <<<

```
;************
;*          *
;*"JUMP.TBL"*
;*          *
;************
;
;
;       THIS FILE CONTAINS THE JUMP TABLE UTILIZED FOR
;       LINKING TOGETHER THE VARIOUS PROGRAM SEGMENTS.
;       ALSO CONTAINED IS THE HEX TABLE CODE USED FOR
;       OUTPUTTING HEX VALUE (IN ASCII) TO THE SCREEN.
;
*=$CF50 ********** HEXIDECIMAL CODE ********
;
A           0123456789ABCDEF
;
*=$CF70 **** COMMAND ROUTINE AND SUBROUTINE JUMP TABLE ****
.      JMP-- $8000  ;PROGRAM START
.      JMP-- $9021  ;DEBUG MONITOR (MAIN DRIVER LOOP)
.      JMP-- $8400  ;INTERNAL TO ASCII HEX OUTPUT
.      JMP-- $8419  ;INTERNAL TO ASCII BINARY OUTPUT
.      JMP-- $8430  ;CLEAR INPUT BUFFER
.      JMP-- $843D  ;INPUT FROM KEYS
.      JMP-- $84F0  ;SET SYSTEM FLAGS
.      JMP-- $8590  ;2/4 CHAR ASCII TO INTERNAL
.      JMP-- $9750  ;NMI SINGLE STEP ROUTINE (TRACE DRIVER)
.      JMP-- $8650  ;DISPLAY REGISTERS
.      JMP-- $88C2  ;BRK EVALUATION DURING NMI
.      JMP-- $8450  ;ERROR/CANCEL RTS
.      JMP-- $845C  ;ERROR/ NORMAL RTS
.      JMP-- $84C1  ;MESSAGE OUTPUT
.      JMP-- $84B7  ;CLEAR BREAK SUBR ENTRY
.      JMP-- $8600  ;ADD TO ZP INDEX 'A'
.      JMP-- $8610  ;ADD TO ZP INDEX 'B'
.      JMP-- $8620  ;ADC TO ZP INDEX 'C'
.      JMP-- $8636  ;COMPARE INDEX A TO B
.      JMP-- $0000  ;SPARE
.      JMP-- $9330  ;DISASSEMBLE ROUTINE
.      JMP-- $9000  ; TEMP
.      JMP-- $9000  ; TEMP
.      JMP-- $8640  ;OUTPUT A SPACE
.      JMP-- $8646  ;OUTPUT A <CR>
.      JMP-- $9000  ;SPARE
.      JMP-- $9000  ;SPARE
.      JMP-- $9000  ;SPARE
.      JMP-- $9000  ;SPARE
.      JMP-- $9000  ;SPARE
.      JMP-- $9000  ;SPARE
.      JMP-- $90A0  ;OUTPUT UNRECOG. CMD
;
```

```
> > >   F I L E :   J U M P . T B L   < < <
                    (CONTINUED)

@$C97E ********** 1 CHARACTER CMD LINK TABLE **********
;       USED BY THE COMMAND INTERPRETER TO LINK
;       TO THE PROPER COMMAND ROUTINES.
;
A           D
T           $E0 $92
A           M
T           $20 $91
A           F
T           $92 $92
A           G
T           $E8 $95
A           L
T           $30 $90
A           S
T           $D0 $90
A           E
T           $B8 $96
A           T
T           $B2 $96
A           X
T           $48 $94
A           P
T           $15 $95
T           $00
;
@$C9AE ********** 2 CHARACTER CMD LINK TABLE **********
;       SAME AS ABOVE EXCEPT FOR TWO CHARACTER COMMANDS.
;
A           B
T           $60 $94
A           D
T           $10 $94
A           S
T           $AA $96
A           C
T           $B0 $94
A           L
T           $08 $96
A           W
T           $02 $96
A           E
T           $B8 $96
A           T
T           $B2 $96
A           H
T           $AA $91
T           $00
```

APPENDIX C

SAMPLE OUTPUT

>>> FILE: SAMPLES.SO <<

```
;
;**************
;*            *
;*"SAMPLES.SO"*
;*            *
;**************
;
;
;     THIS FILE CONTAINS VARIOUS SAMPLE ROUTINES FROM
.     WHICH THE FOLLOWING OUTPUT WAS GENERATED.
;
;
@$8000 ********** SAMPLE #1 **********
;
;       THIS SAMPLE IS USED FOR
;         - GENERAL SINGLE STEPPING.
:         - WATCH MICROPROCESSOR REGISTER.
;         - WATCH MEMORY.
:
.     LDA#  $01      ;LOAD ACC WITH $01
.     LDY#  $03      ;LOAD Y-REG WITH $03
*1    STY-- $6000    ;STORE Y-REG AT $6000
.     ADC-- $6000    ;ADD VALUE AT $6000 ACC
.     DEY            ;DECREMENT Y-REG
.     BNE   *1       ;IF NOT ZERO GOTO *1
.     BRK            ;BREAK AND END ROUTINE
;
;
@$8020 ********** SAMPLE #2 **********
;
;       THIS SAMPLE IS USED FOR
;         -SINGLE STEPPING WITH A SUBROUTINE.
;         -SINGLE STEPPING WITH A SUBROUTINE AND THE
;          SUBROUTINE SUPPRESSION PARAMETER ACTIVE.
;
.     LDA#  $01      ;LOAD ADD WITH $01
.     TAY            ;TRANSFER ACC TO Y-REG
.     STY-- $6000    ;STORE Y-REG AT $6000
.     LDX-  $19      ;LOAD X-REG FROM ZERO PAGE $19
.     LDY-- $C010    ;LOAD Y-REG FROM $C010
.     JSR-- $8040    ;JUMP TO SUBROUTINE AT $8040
.     TAY            ;TRANSFER ACC TO Y-REG
.     INX            ;INCREMENT X-REG
.     INX            ;INCREMENT X-REG
.     STX-- $6001    ;STORE X-REG AT $6001
.     BRK            ;BREAK AND END ROUTINE
;
@$8040 SAMPLE SUBROUTINE
;
.     STY-- $6001    ;STORE Y-REG AT $6001
```

>>> FILE: SAMPLES.SO <<

(CONTINUED)

```
.     INY            ;INCREMENT Y-REG
.     TYA            ;TRANSFER Y-REG TO ACC
.     ADC#  $3F      ;ADD $3F TO ACC
.     STA-- $6002    ;STORE ACC AT $6002
.     RTS            ;RETURN TO CALLING PROGRAM
:
;
@$2000 ********** SAMPLE #3 **********
;
;       THIS SAMPLE IS USED FOR
;         - SINGLE AND MULTIPLE EXECUTION OF
;           BREAK POINTS.
;
.     LDX#  $00      ;LOAD X-REG WITH $00
.     LDY#  $FF      ;LOAD Y-REG WITH $FF
*2    INX            ;INCREMENT X-REG
.     DEY            ;DECREMENT Y-REG
.     JMP-- *2       ;JUMP TO *2
```

```
>P
BREAKPOINT #1 NOT SET
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH  NOT SET
>TC
  PC    MEMONIC     ACC XR YR SP NV*BDIZC
 8000 LDA#  $01      01 00 00 F6 00100000
 8002 LDY#  $03      01 00 03 F6 00100000
 8004 STY-- $6000    01 00 03 F6 00100000
 8007 ADC-- $6000    04 00 03 F6 00100000
 800A DEY            04 00 02 F6 00100000
 800B BNE   $8004    04 00 02 F6 00100000
 8004 STY-- $6000    04 00 02 F6 00100000
 8007 ADC-- $6000    06 00 02 F6 00100000
 800A DEY            06 00 01 F6 00100000
 800B BNE   $8004    06 00 01 F6 00100000
 8004 STY-- $6000    06 00 01 F6 00100000
 8007 ADC-- $6000    07 00 01 F6 00100000
 800A DEY            07 00 00 F6 00100010
 800B BNE   $8004    07 00 00 F6 00100010
                    ACC XR YR SP NV*BDIZC
BREAK AT $800D       07 00 00 F6 00110010
>
```

This exaple shows normal single stepping. It utilizes sample #1. The command issued was "Trace Continuously", TC.

```
>P
BREAKPOINT #1 NOT SET
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH  NOT SET
>T 01
  PC   MEMONIC      ACC XR YR SP NV*BDIZC
 8000 LDA#  #01      01 06 01 F7 00100000
>T 02
  PC   MEMONIC      ACC XR YR SP NV*BDIZC
 8002 LDY#  #03      01 06 03 F7 00100000
 8004 STY-- $6000    01 06 03 F7 00100000
>T 03
  PC   MEMONIC      ACC XR YR SP NV*BDIZC
 8007 ADC-- $6000    04 06 03 F7 00100000
 800A DEY            04 06 02 F7 00100000
 800B BNE   #8004    04 06 02 F7 00100000
>T 04
  PC   MEMONIC      ACC XR YR SP NV*BDIZC
 8004 STY-- $6000    04 06 02 F7 00100000
 8007 ADC-- $6000    06 06 02 F7 00100000
 800A DEY            06 06 01 F7 00100000
 800B BNE   #8004    06 06 01 F7 00100000
>
```

This example utilizes the same code as the previously sample. In this case, the commands "Trace 01", "Trace 02", "Trace 03", and "Trace 04" were issued. This caused single stepping of only the specified number of lines (specified in hexidecimal).

```
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH  NOT SET
>TC
  PC    MEMONIC       ACC XR  YR  SP  NV*B01ZC
 8020 LDA#   $01       01  01  E9  F6  00100000
 8022 TAY              01  01  01  F6  00100000
 8023 STY--  $6000     01  01  01  F6  00100000
 8026 LDX-   $19       01  FF  01  F6  10100000
 8028 LDY--  $C010     01  FF  A9  F6  10100000
 802B JSR--  $8040     01  FF  A9  F4  10100000
 8040 STY--  $6001     01  FF  A9  F4  10100000
 8043 INY              01  FF  AA  F4  10100000
 8044 TYA              AA  FF  AA  F4  10100000
 8045 ADC#   $3F       E9  FF  AA  F4  10100000
 8047 STA--  $6002     E9  FF  AA  F4  10100000
 804A RTS              E9  FF  AA  F6  10100000
 802E TAY              E9  FF  E9  F6  10100000
 802F INX              E9  00  E9  F6  00100010
 8030 INX              E9  01  E9  F6  00100000
 8031 STX--  $6001     E9  01  E9  F6  00100000
                      ACC  XR  YR  SP  NV*B01ZC
BREAK AT #8034         E9  01  E9  F6  00110000
>


BREAKPOINT #1 NOT SET
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION SET
WATCH  NOT SET
>TC
  PC    MEMONIC       ACC XR  YR  SP  NV*B01ZC
 8020 LDA#   $01       01  01  E9  F6  00100000
 8022 TAY              01  01  01  F6  00100000
 8023 STY--  $6000     01  01  01  F6  00100000
 8026 LDX-   $19       01  FF  01  F6  10100000
 8028 LDY--  $C010     01  FF  A9  F6  10100000
 802B JSR--  $8040     01  FF  A9  F4  10100000
                       E9  FF  AA  F6  10100000
 802E TAY              E9  FF  E9  F6  10100000
 802F INX              E9  00  E9  F6  00100010
 8030 INX              E9  01  E9  F6  00100000
 8031 STX--  $6001     E9  01  E9  F6  00100000
                      ACC  XR  YR  SP  NV*B01ZC
BREAK AT #8034         E9  01  E9  F6  00110000
>
```

The above examples show normal execution of a subroutine and execution of a subroutine with the "Subroutine Suppression" parameter active. As seen, at address $802B (top example) a subroutine is entered. The subroutine is exitted at address $802E. The bottem example shows that the subroutine is not displayed when "subroutine Suppression is active. Only the first line of it (the JSR call itself) and the register values upon RTS. These examples utilize sample #2.

```
>P
BREAKPOINT #1 NOT SET
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH Y FOR $01
>TC
  PC    MEMONIC     ACC XR YR SP NV*BDIZC
 8000 LDA#  $01      01 00 00 FE 00100000
 8002 LDY#  $03      01 00 03 FE 00100000
 8004 STY-- $6200    01 00 03 FE 00100000
 8007 ADC-- $6000    04 00 03 FE 00100000
 800A DEY            04 00 02 FE 00100000
 800B BNE   $8004    04 00 02 FE 00100000
 8004 STY-- $6200    04 00 02 FE 00100000
 8007 ADC-- $6000    05 00 02 FE 00100000
 800A DEY            06 00 01 FE 00100000
WATCH  COMPLETE
```

This example shows the watch command. It utilizes sample #1. The watch parameter has been set to "Watch Y-reg for $01". Upon issuance of the "Trace continuously" command, execution starts. Execution is halted when the Y-reg attains the value $01.

```
BREAKPOINT #1 NOT SET
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH X FOR $05
>TC
  PC    MEMONIC      ACC XR YR SP NV*BDIZC
 8060 LDX#   $00      00 00 FF F6 01100011
 8062 LDY#   $FF      00 00 FF F6 11100001
 8064 INX             00 01 FF F6 01100001
 8065 DEY             00 01 FE F6 11100001
 8066 JMP-- $8064     00 01 FE F6 11100001
 8064 INX             00 02 FE F6 01100001
 8065 DEY             00 02 FD F6 11100001
 8066 JMP-- $8064     00 02 FD F6 11100001
 8064 INX             00 03 FD F6 01100001
 8065 DEY             00 03 FC F6 11100001
 8066 JMP-- $8064     00 03 FC F6 11100001
 8064 INX             00 04 FC F6 01100001
 8065 DEY             00 04 FB F6 11100001
 8066 JMP-- $8064     00 04 FB F6 11100001
 8064 INX             00 05 FB F6 01100001
WATCH  COMPLETE
 >
```

This example show the watch command utilizing sample #3. The parameter "Watch X-reg for $05" was issued. The "Trace continuously" command begins execution. When the X-register attains the value $05, execution is halted.

```
BREAKPOINT #1 NOT SET
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH $6000 FOR $03
>TC
  PC   MEMONIC     ACC XR YR SP NV*BDIZC
 8000 LDA#  $01     01 05 FB F6 01100001
 8002 LDY#  $03     01 05 03 F6 01100001
 8004 STY-- $6000   01 05 03 F6 01100001
WATCH  COMPLETE
>
```

This is another example of the watch command utilizing sample #1. The parameter "Watch memory address $6000 for $03" was issued. Again the "Trace continuously" command was issued. When the value at address $6000 contained $03, execution was halted.

```
BREAKPOINT #1 SET AT $8065
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH  NOT SET
>TC
  PC    MEMONIC      ACC XR YR SP NV*BDIZC
 8060 LDX#  $00      0E 00 FC F7 00100010
 8062 LDY#  $FF      0E 00 FF F7 10100000
 8064 INX            0E 01 FF F7 00100000
                     ACC XR YR SP NV*BDIZC
BREAK AT $8065       0E 01 FF F7 00110000
>

BREAKPOINT #1 SET AT $8065
BREAKPOINT #2 NOT SET
SUBROUTINE SUPRESSION NOT SET
WATCH  NOT SET
>TB 04
  PC    MEMONIC      ACC XR YR SP NV*BDIZC
 8060 LDX#  $00      0E 00 FC F7 00100010
 8062 LDY#  $FF      0E 00 FF F7 10100000
 8064 INX            0E 01 FF F7 00100000
+8065 DEY            0E 01 FE F7 10100000
 8066 JMP-- $8064    0E 01 FE F7 10100000
 8064 INX            0E 02 FE F7 00100000
+8065 DEY            0E 02 FD F7 10100000
 8066 JMP-- $8064    0E 02 FD F7 10100000
 8064 INX            0E 03 FD F7 00100000
+8065 DEY            0E 03 FC F7 10100000
 8066 JMP-- $8064    0E 03 FC F7 10100000
 8064 INX            0E 04 FC F7 00100000
                     ACC XR YR SP NV*BDIZC
BREAK AT $8065       0E 04 FC F7 00110000
```

Utilizing sample #3, the above examples show both normal break points and the multiple detection of break points. In both examples, the break point is set at $8065 (break point #1). In the top case, when address $8065 is reached, the break point is activated. In the lower case, the command "Trace until Break $04" was issued. In this case, the break point is executed through itself three times (indicated by the "+"'s on the output) before finally halting execution on the fourth detection.

```
BREAKPOINT #1 SET AT $8064
BREAKPOINT #2 SET AT $8065
SUBROUTINE SUPRESSION NOT SET
WATCH  NOT SET
>TB 0A
  PC   MEMONIC     ACC XR YR SP NV*BOIZC
 8060 LDX#  $00     01 00 F6 F6 01100011
 8062 LDY#  $FF     01 00 FF F6 11100001
+8064 INX           01 01 FF F6 01100001
+8065 DEY           01 01 FE F6 11100001
 8066 JMP-- $8064   01 01 FE F6 11100001
+8064 INX           01 02 FE F6 01100001
+8065 DEY           01 02 FD F6 11100001
 8066 JMP-- $8064   01 02 FD F6 11100001
+8064 INX           01 03 FD F6 01100001
+8065 DEY           01 03 FC F6 11100001
 8066 JMP-- $8064   01 03 FC F6 11100001
+8064 INX           01 04 FC F6 01100001
+8065 DEY           01 04 FB F6 11100001
 8066 JMP-- $8064   01 04 FB F6 11100001
+8064 INX           01 05 FB F6 01100001
                   ACC XR YR SP NV*BOIZC
BREAK AT $8065      01 05 FB F6 01110001
>
```

The above example, utilizing sample #3, shows two break points set. One is set at $8064, and one at $8065. The command "Trace until Break $0A" was issued. As seen, the two break points were encountered nine times. One the tenth detection, execution was halted.

VITA

The author, Charles Raymond Jacobs, was born in Chicago, Illinois on December 16, 1960. He was the youngest of two children born to Charles E. Jacobs and Evelyn A. Jacobs (nee Wilferling).

His educational background consists of: Saint Pascals Elementary School (Chicago), DeSales High School (Louisville, Kentucky), and The University of Louisville, Speed Scientific School (Louisville). He received his Bachelors of Science Degree with Concentration in Electrical Engineering in May, 1984. In February, 1985 he began full time work at Louisville Gas and Electric Company while continuing his education toward his Master of Engineering Degree. The degree will be awarded August, 1987.